CT4101

Machine Learning

Name: Andrew Hayes
E-mail: `a.hayes18@universityofgalway.ie`
Student ID: 21321503

2024–11–27

Contents

1 Introduction

1.1 Lecturer Contact Details

- Dr. Frank Glavin.
- `frank.glavin@universityofgalway.ie`

1.2 Grading

- Continuous Assessment: 30% (2 assignments, worth 15% each).
- Written Exam: 70% (Last 2 year's exam papers most relevant).

There will be no code on the exam, but perhaps pseudo-code or worked mathematical examples.

1.3 Module Overview

Machine Learning (ML) allows computer programs to improve their performance with experience (i.e., data). This module is targeted at learners with no prior ML experience, but with university experience of mathematics & statistics and **strong** programming skills. The focus of this module is on practical applications of commonly used ML algorithms, including deep learning applied to computer vision. Students will learn to use modern ML frameworks (e.g., scikit-learn, Tensorflow / Keras) to train & evaluate models for common categories of ML task including classification, clustering, & regression.

1.3.1 Learning Objectives

On successful completion, a student should be able to:

1. Explain the details of commonly used Machine Learning algorithms.
2. Apply modern frameworks to develop models for common categories of Machine Learning task, including classification, clustering, & regression.
3. Understand how Deep Learning can be applied to computer vision tasks.
4. Pre-process datasets for Machine Learning tasks using techniques such as normalisation & feature selection.
5. Select appropriate algorithms & evaluation metrics for a given dataset & task.
6. Choose appropriate hyperparameters for a range of Machine Learning algorithms.
7. Evaluate & interpret the results produced by Machine Learning models.
8. Diagnose & address commonly encountered problems with Machine Learning models.
9. Discuss ethical issues & emerging trends in Machine Learning.

2 What is Machine Learning?

There are many possible definitions for "machine learning":

- Samuel, 1959: "Field of study that gives computers the ability to learn without being explicitly programmed".
- Witten & Frank, 1999: "Learning is changing behaviour in a way that makes *performance* better in the future".
- Mitchelll, 1997: "Improvement with experience at some task". A well-defined ML problem will improve over task T with regards to **performance** measure P, based on experience E.
- Artificial Intelligence $\neq$ Machine Learning $\neq$ Deep Learning.

- Artificial Intelligence $\nsupseteq$ Machine Learning $\nsupseteq$ Deep Learning.

Machine Learning techniques include:

- Supervised learning.
- Unsupervised learning.
- Semi-Supervised learning.
- Reinforcement learning.

Major types of ML task include:

1. Classification.
2. Regression.
3. Clustering.
4. Co-Training.
5. Relationship discovery.
6. Reinforcement learning.

Techniques for these tasks include:

1. **Supervised learning:**
    - **Classification:** decision trees, SVMs.
    - **Regression:** linear regression, neural nets, k-NN (good for classification too).
2. **Unsupervised learning:**
    - **Clustering:** k-Means, EM-clustering.
    - **Relationship discovery:** association rules, bayesian nets.
3. **Semi-Supervised learning:**
    - **Learning from part-labelled data:** co-training, transductive learning (combines ideas from clustering & classification).
4. **Reward-Based:**
    - **Reinforcement learning:** Q-learning, SARSA.

In all cases, the machine searches for a **hypothesis** that best describes the data presented to it. Choices to be made include:

- How is the hypothesis expressed? e.g., mathematical equation, logic rules, diagrammatic form, table, parameters of a model (e.g. weights of an ANN), etc.
- How is search carried out? e.g., systematic (breadth-first or depth-first) or heuristic (most promising first).
- How do we measure the quality of a hypothesis?
- What is an appropriate format for the data?
- How much data is required?

To apply ML, we need to know:

- How to formulate a problem.
- How to prepare the data.
- How to select an appropriate algorithm.
- How to interpret the results.

To evaluate results & compare methods, we need to know:

- The separation between training, testing, & validation.
- Performance measures such as simple metrics, statistical tests, & graphical methods.
- How to improve performance.
- Ensemble methods.
- Theoretical bounds on performance.

2.1 Data Mining

Data Mining is the process of extracting interesting knowledge from large, unstructured datasets. This knowledge is typically non-obvious, comprehensible, meaningful, & useful.

The storage "law" states that storage capacity doubles every year, faster than Moore's "law", which may results in write-only "data tombs". Therefore, developments in ML may be essential to be able to process & exploit this lost data.

2.2 Big Data

Big Data consists of datasets of scale & complexity such that they can be difficult to process using current standard methods. The data scale dimensions are affected by one or more of the "3 Vs":

- **Volume:** terabytes & up.
- **Velocity:** from batch to streaming data.
- **Variety:** numeric, video, sensor, unstructured text, etc.

It is also fashionable to add more "Vs" that are not key:

- **Veracity:** quality & uncertainty associated with items.
- **Variability:** change / inconsistency over time.
- **Value:** for the organisation.

Key techniques for handling big data include: sampling, inductive learning, clustering, associations, & distributed programming methods.

3 Introduction to Python

Python is a general-purpose high-level programming language, first created by Guido van Rossum in 1991. Python programs are interpreted by an *interpreter*, e.g. **CPython** – the reference implementation supported by the Python Software Foundation. CPython is both a compiler and an interpreter as it first compiles Python code into bytecode before interpreting it.

Python interpreters are available for a wide variety of operating systems & platforms. Python supports multiple programming paradigms, including procedural programming, object-oriented programming, & functional programming. Python is **dynamically typed**, unlike languages such as C, C++, & Java which are *statically typed*, meaning that many common error checks are deferred until runtime in Python, whereas in a statically typed language like Java these checks are performed during compilation.

Python uses **garbage collection**, meaning that memory management is handled automatically and there is no need for the programmer to manually allocate & de-allocate chunks of memory.

Python is used for all kinds of computational tasks, including:

- Scientific computing.
- Data analytics.
- Artificial Intelligence & Machine Learning.
- Computer vision.
- Web development / web apps.
- Mobile applications.
- Desktop GUI applications.

While having relatively simple syntax and being easy to learn for beginners, Python also has very advanced functionality. It is one of the most widely used programming languages, being both open source & freely available. Python programs will run almost anywhere that there is an installation of the Python interpreter. In contrast, many languages such as C or C++ have separate binaries that must be compiled for each specific platform & operating system.

Python has a wide array of libraries available, most of which are free & open source. Python programs are usually much shorter than the equivalent Java or C++ code, meaning less code to write and faster development times for experienced Python developers. Its brevity also means that the code is easier to maintain, debug, & refactor as much less source code is required to be read for these tasks. Python code can also be run without the need for ahead-of-time compilation (as in C or C++), allowing for faster iterations over code versions & faster testing. Python can also be easily extended & integrated with software written in many other programming languages.

Drawbacks of using Python include:

- **Efficiency:** Program execution speed in Python is typically a lot slower than more low-level languages such as C or C++. The relative execution speed of Python compared to C or C++ depends a lot on coding practices and the specific application being considered.
- **Memory Management** in Python is less efficient than well-written C or C++ code although these efficiency concerns are not usually a major issues, as compute power & memory are now relatively cheap on desktop, laptop, & server systems. Python is used in the backend of large web services such as Spotify & Instagram, and performs adequately. However, these performance concerns may mean that Python is unsuitable for some performance-critical applications, e.g. resource-intensive scientific computing, embedded devices, automotive, etc. Faster alternative Python implementations such as **PyPy** are also available, with PyPy providing an average of a four-fold speedup by implementing advanced compilation techniques. It's also possible to call code that is implemented in C within Python to speed up performance-critical sections of your program.
- **Dynamic typing** can make code more difficult to write & debug compared to statically-typed languages, wherein the compiler checks that all variable types match before the code is executed.
- **Python2 vs Python3:** There are two major version of Python in widespread use that are not compatible with each other due to several changes that were made when Python3 was introduced. This means that some libraries that were originally written in Python2 have not been ported over to Python3. Python2 is now mostly used only in legacy business applications, while most new development is in Python3. Python2 is no longer supported or receives updates as of 2020.

3.1 Running Python Programs

Python programs can be executed in a variety of different ways:

- through the Python interactive shell on your local machine.
- through remote Python interactive shells that are accessible through web browsers.
- by using the console of your operating system to launch a standalone Python script (`.py` file).
- by using an IDE to launch a `.py` file.
- as GUI applications using libraries such as Tkinter PyQt.
- as web applications that provide services to other computers, e.g. by using the Flask framework to create a web server with content that can be accessed using web browsers.
- through Jupyter / JupyterLab notebooks, either hosted locally on your machine or cloud-based Jupyter notebook execution environments such as Google Colab, Microsoft Azure Notebooks, Binder, etc.

3.2 Hello World

The following programs writes "Hello World!" to the screen.

```python
print("Hello World!")
```

Listing 1: `helloworld.py`

3.3 PEP 8 Style Guide

PEPs (Python Enhancement Proposals) describe & document the way in which the Python language evolves over time, e.g. addition of new features. Backwards compatibility policy etc. PEPSs can be proposed, then accepted or rejected. The full list is available at `https://www.python.org/dev/peps/`. **PEP 8** gives coding conventions for the Python code comprising the standard library in the main Python distribution. See: `https://www.python.org/dev/peps/pep-0008/`. It contains conventions for the user-defined names (e.g., variables, functions, packages), as well as code layout, line length, use of blank lines, style of comments, etc.

Many professional Python developers & companies adhere to (at least some of) the PEP8 conventions. It is important to learn to follow these conventions from the start, especially if you want to work with other programmers, as experienced Python developers will often flag violations of the PEP 8 conventions during code reviews. Of course, many companies & open-source software projects have defined their own internal coding style guidelines which take precedence over PEP 8 in the case of conflicts. Following PEP 8 conventions is relatively easy if you are using a good IDE, e.g. PyCharm automatically finds & alerts you to violations of the PEP 8 conventions.

3.3.1 Variable Naming Conventions

According to PEP 8, variable names "should be lowercase, with words separated by underscores as necessary to improve readability", i.e. `snake_case`. "Never use the characters `l`, `O`, or `I` as single-character variable names. In some fonts, these characters are indistinguishable from the numerals one & zero. When tempted to use `l`, use `L` instead". According to PEP 8, different naming conventions are used for different identifiers, e.g.: "Class names should normally use the CapWords convention". This helps programmers to quickly & easily distinguish which category an identifier name represents.

3.3.2 Whitespace in Python

A key difference between Python and other languages such as C is that whitespace has meaning in Python. The PEP 8 style guidelines say to "Use 4 spaces per indentation level", not 2 spaces, and not a tab character. This applies to all indented code blocks.

3.4 Dynamic Typing

In Python, variable names can point to objects of any type. Built-in data types in python include **`str`**, **`int`**, **`float`**, etc. Each type can hold a different type of data. Because variables in Python are simply pointers to objects, the variable names themselves do not have any attached type information. Types are linked not to the variable names but to the objects themselves.

```
x = 4
print(type(x))  # prints "<class 'int'>" to the console
x = "Hello World!"
print(type(x))  # prints "<class 'str'>" to the console
x = 3.14159
print(type(x))  # prints "<class 'float'>" to the console
```

Listing 2: Dynamic Typing Example

Note that `type()` is a built-in function that returns the type of any object that is passed to it as an argument. It returns a **type object**.

Because the type of object referred to by a variable is not known until runtime, we say that Python is a **dynamically typed language**. In **statically typed languages**, we must declare the type of a variable before it is used: the type of every variable is known before runtime.

Another important difference between Python and statically typed languages is that we do not need to declare variables before we use them. Assigning a value to a previously undeclared variable name is fine in Python.

3.5 Modules, Packages, & Virtual Environments

3.5.1 Modules

A **module** is an object that serves as an organisational unit of Python code. Modules have a *namespace* containing arbitrary Python objects and are loaded into Python by the process of *importing*. A module is essentially a file containing Python definitions & statements.

Modules can be run either as standalone scripts or they can be **imported** into other modules so that their built-in variables, functions, classes, etc. can be used. Typically, modules group together statements, functions, classes, etc. with related functionality. When developing larger programs, it is convenient to split the source code up into separate modules. As well as creating our own modules to break up our source code into smaller units, we can also import built-in modules that come with Python, as well as modules developed by third parties.

Python provides a comprehensive set of built-in modules for commonly used functionality, e.g. mathematical functions, date & tie, error handling, random number generation, handling command-line arguments, parallel processing, networking, sending email messages, etc. Examples of modules that are built-in to Python include `math`, `string`, `argparse`, `calendar`, etc. The `math` module is one of the most commonly used modules in Python, although the functions in the `math` module do not support complex numbers; if you require complex number support, you can use the `cmath` module. A full list of built-in modules is available at `https://docs.python.org/3/py-modindex.html`.

3.5.2 Packages

Packages are a way of structuring Python's module namespace by using "dotted module names": for example, the module name `A.B` designates a submodule named `B` in a package `A`. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like `NumPy` or `Pillow` from having to worry about each other's module names. Individual modules can be imported from a package: `import sound.effects.echo`.

PEP 8 states that "Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability. Python packages should also have short, all-lowercase names, although the use of underscores is discouraged."

3.5.3 Managing Packages with `pip`

`pip` can be used to install, upgrade, & remove packages and is supplied by default with your Python installation. By default, `pip` will install packages from the Python Package Index (PyPI) `https://pypi.org`. You can browse the Python Package Index by visiting it in your web browser. To install packages from PyPI:

```
python -m pip install projectname
```

To upgrade a package to the latest version:

```
python -m pip install --upgrade projectname
```

3.5.4 Virtual Environments

Python applications will often use packages & modules that don't come as part of the standard library. Applications will sometimes need a specific version of a library, because the application may require that a particular bug has been fixed or the application may have been written using an obsolete version of the library's interface. This means that it may not be possible for one Python installation to meet the requirements of every application. If application A needs version 1.0 of a particular module but application B needs version 2.0, then the requirements are in conflict and installing either version 1.0 or 2.0 will leave one application unable to run. The solution for this problem is to create a **virtual environment**, a self-contained directory tree that contains a Python installation for a particular version of Python plus a number of additional packages. Different applications can then use different virtual environments.

By default, most IDEs will create a new virtual environment for each new project created. It is also possible to set up a project to run on a specific pre-configured virtual environment. The built-in module `venv` can also be used to create & manage virtual environments through the console.

To use the `venv` module, first decide where you want the virtual environment to be created, then open a command line at that location use the command `python -m venv environmentname` to create a virtual environment with the specified name. You should then see the directory containing the virtual environment appear on the file system, which can then be activated using the command `source environmentname/bin/activate`.

To install a package to a virtual environment, first activate the virtual environment that you plan to install it to and then enter the command `python -m pip install packagename`.

If you have installed packages to a virtual environment, you will need to make that virtual environment available to Jupyter Lab so that your `.ipynb` files can be executed on the correct environment. You can use the package `ipykrenel` to do this.

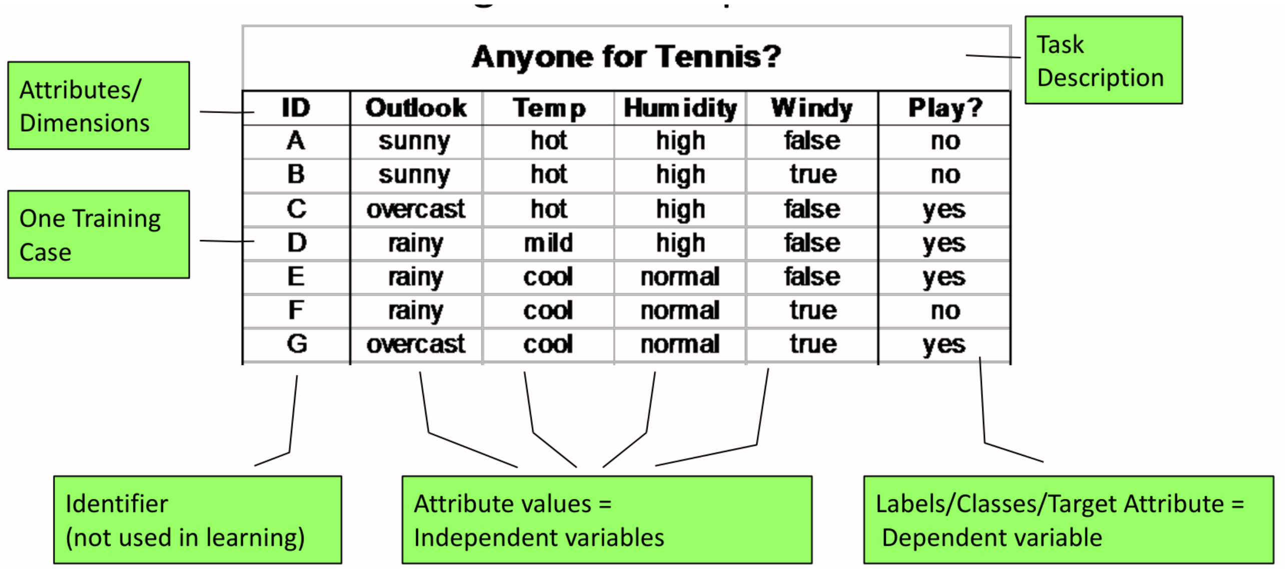
4 Classification

4.1 Supervised Learning Principles

Recall from before that there are several main types of machine learning techniques, including **supervised learning**, unsupervised learning, semi-supervised learning, & reinforcement learning. Supervised learning tasks include both **classification** & regression.

The task definition of supervised learning is to, given examples, return a function h (hypothesis) that approximates some "true" function f that (hypothetically) generated the labels for the examples. We need to have a set of examples called the **training data**, each having a **label** & a set of **attributes** that have known **values**.

We consider the labels (classes) to be the outputs of some function f: the observed attributes are its inputs. We denote the attribute value inputs x and labels are their corresponding outputs $f(x)$. An example is a pair $(x, f(x))$. The function f is unknown, and we want to discover an approximation of it h. We can then use h to predict labels of new data (generalisation). This is also known as **pure inductive learning**.



Anyone for Tennis?

ID	Outlook	Temp	Humidity	Windy	Play?
A	sunny	hot	high	false	no
B	sunny	hot	high	true	no
C	overcast	hot	high	false	yes
D	rainy	mild	high	false	yes
E	rainy	cool	normal	false	yes
F	rainy	cool	normal	true	no
G	overcast	cool	normal	true	yes

Task Description
Attributes/ Dimensions
One Training Case
Identifier (not used in learning)
Attribute values = Independent variables
Labels/Classes/Target Attribute = Dependent variable

Figure 1: Training Data Example for a Classification Task

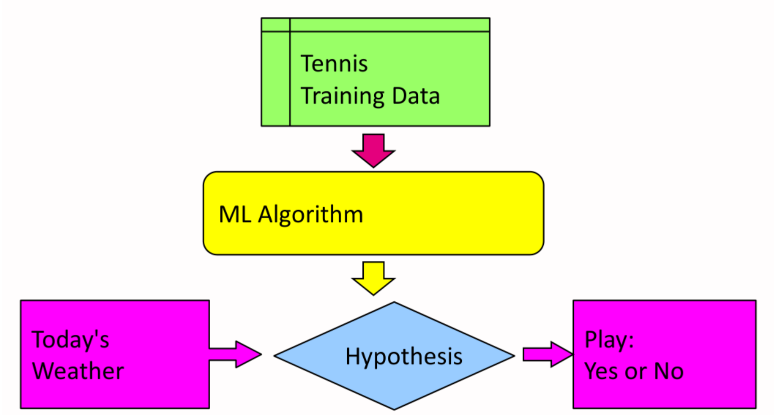


Figure 2: Overview of the Supervised Learning Process

4.2 Introduction to Classification

The simplest type of classification task is where instances are assigned to one of two categories: this is referred to as a **binary classification task** or two-class classification task. Many popular machine learning problems fall into this category:

- Is cancer present in a scan? (Yes / No).
- Should this loan be approved? (Yes / No).
- Sentiment analysis in text reviews of products (Positive / Negative).
- Face detection in images (Present / Not Present).

The more general form of classification task is the **mutli-class classification** where the number of classes is ≥ 3.

4.2.1 Example Binary Classification Task

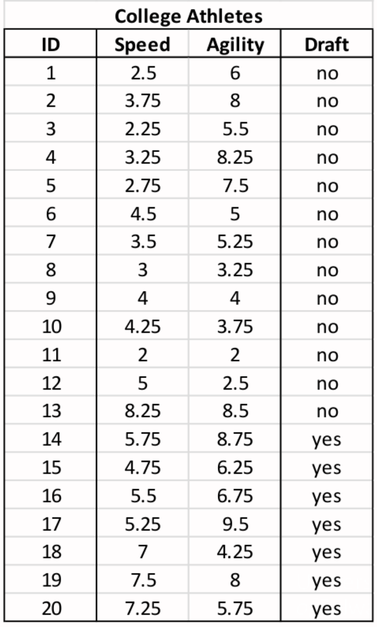
Objective: build a binary classifier to predict whether a new previously unknown athlete who did not feature in the dataset should be drafted.

There are 20 examples in the dataset, see `college_athletes.csv` on Canvas.

The college athlete's dataset contains two attributes:

- Speed (continuous variable).
- Agility (continuous variable).

The target data: whether or not each athlete was drafted to a professional team (yes / no).



College Athletes

ID	Speed	Agility	Draft
1	2.5	6	no
2	3.75	8	no
3	2.25	5.5	no
4	3.25	8.25	no
5	2.75	7.5	no
6	4.5	5	no
7	3.5	5.25	no
8	3	3.25	no
9	4	4	no
10	4.25	3.75	no
11	2	2	no
12	5	2.5	no
13	8.25	8.5	no
14	5.75	8.75	yes
15	4.75	6.25	yes
16	5.5	6.75	yes
17	5.25	9.5	yes
18	7	4.25	yes
19	7.5	8	yes
20	7.25	5.75	yes

Figure 3: Example Dataset for a Binary Classification Task

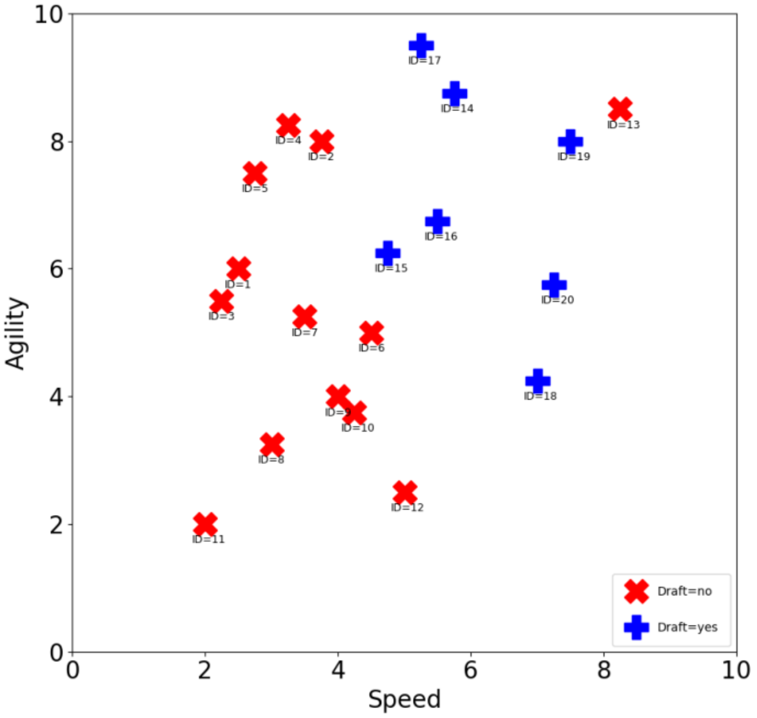


Figure 4: Feature Space Plot for the College Athlete's Dataset

We want to decide on a reasonable **decision boundary** to categorise new unseen examples, such as the one denoted by the purple question mark below. We need algorithms that will generate a hypothesis / model consistent with the training data. Is the decision boundary shown below in thin black lines a good one? It is consistent with all of the training data, but it was drawn manually; in general, it won't be possible to manually draw such decision boundaries when dealing with higher dimensional data (e.g., more than 3 features).

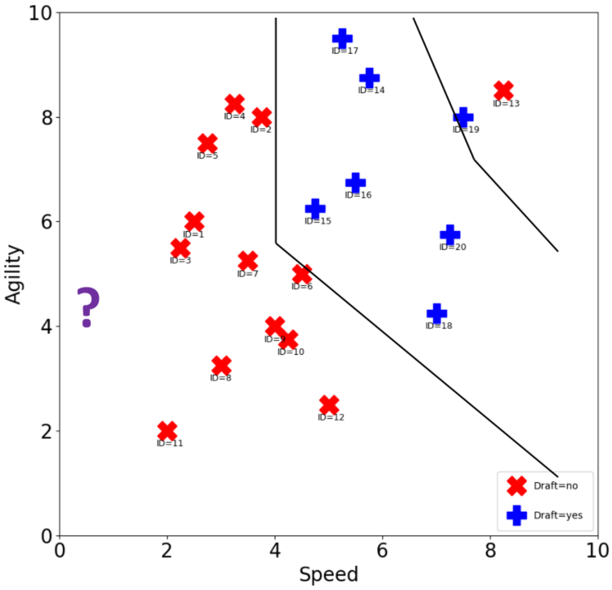


Figure 5: Feature Space Plot for the College Athlete's Dataset

4.2.2 Example Classification Algorithms

There are many machine learning algorithms available to learn a classification hypothesis / model. Some examples (with corresponding scikit-learn classes) are:

- k-nearest neighbours: scikit-learn `KNeighboursClassifier`.
- Decision trees: scikit-learn `DecisionTreeClassifier`.
- Gaussian Processes: scikit-learn `GaussianProcessClassifier`.
- Neural networks: scikit-learn `MLPClassifier`.
- Logistic regression: scikit-learn `LogisticRegression`. Note that despite its name, logistic regression is a linear model for classification rather than regression.

4.2.3 Logistic Regression on the College Athletes Dataset

Below is an example of a very simple hypothesis generated using an ML model – a linear classifier created using the scikit-learn `LogisticRegression` with the default settings.

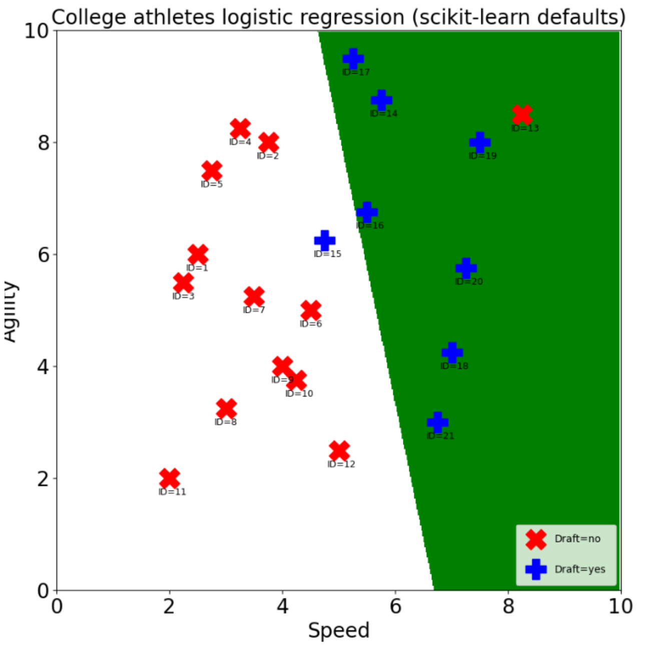


Figure 6: Logistic Regression on the College Athletes Dataset

Is this a good decision boundary? $\frac{19}{21}$ training examples correct = 90.4% accuracy. Note how the decision boundary is a straight line (in 2D). Note also that using logistic regression makes a strong underlying assumption that the data is **linearly separable**.

4.2.4 Decision Tree on the College Athletes Dataset

Below is an example of a more complex hypothesis, generated using the scikit-learn `DecisionTreeClassifier` with the default settings.

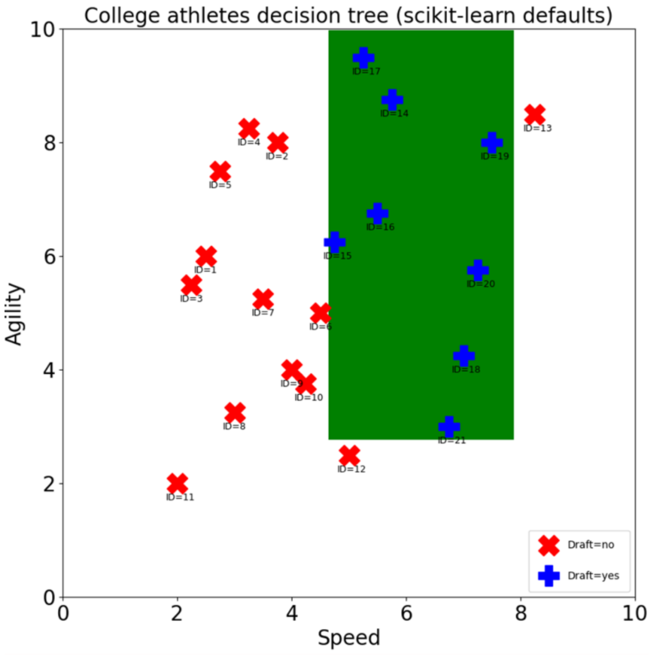


Figure 7: Decision Tree on the College Athletes Dataset

Note the two linear decision boundaries: this is a very different form of hypothesis compared to logistic regression. Is this a good decision boundary? $\frac{21}{21}$ training examples correct = 100% accuracy.

4.2.5 Gaussian Process on the College Athletes Dataset

Below is an example of a much more complex hypothesis generated using the scikit-learn `GaussianProcessClassifier` with the default settings.

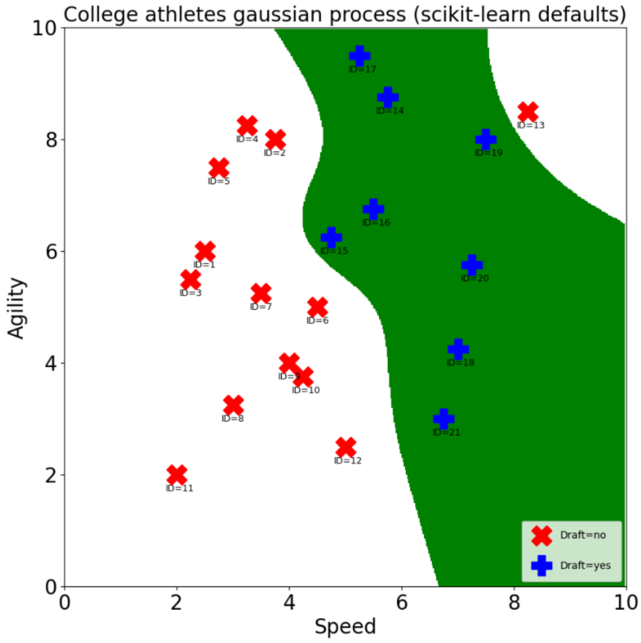


Figure 8: Gaussian Process on the College Athletes Dataset

Note the smoothness of the decision boundary compared to the other methods. Is this a good decision boundary? $\frac{21}{21}$ training examples correct = 100% accuracy.

Which of the three models explored should we choose? It's complicated; we need to consider factors such as accuracy of the training data & independent test data, complexity of the hypothesis, per-class accuracy etc.

4.2.6 Use of Independent Test Data

Use of separate training & test datasets is very important when developing an ML model. If you use all of your data for training, your model could potentially have good performance on the training data but poor performance on new independent test data.

5 k-Nearest Neighbours Algorithm

k-**nearest neigbours** (or k-NN) is one of the simplest machine learning algorithms. It generates a hypothesis using a very simple principle: predictions for the label or value assigned to a *query instance* should be made based on the most *similar* instances in the training dataset. Hence, this is also known as **similarity-based learning**.

k-NN can be used for both classificatoin & regression tasks, although for now we will focus only on its application to classification tasks using the scikit-learn implementation `KNeighborsClassifier`.

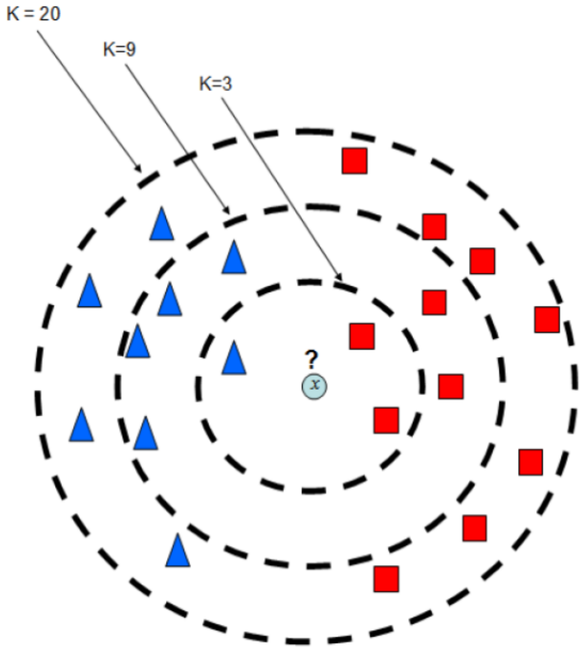


Figure 9: K-Nearest Neighbour Example

The operation of the k-NN algorithm is relatively easy to appreciate. The key insight is that each example is a point in the feature space. If samples are close to each other in the feature space, they should be close in their target values. This is related to *code-based reasoning*. When you want to classify a new **query case**, you compare it to the stored set and retrieve the k most similar instances. The query case is the given a label based on the most similar instances.

The prediction for a query case is based on several (k) nearest neighbours. We compute the similarity of the query case to all stored cases, and pick the nearest k neighbours; the simplest way to do this is to sort the instances by distance and pick the lowest k instances. A more efficient way of doing this would be to identify the k nearest instances in a single pass through the list of distances. The k nearest neighbours then vote on the classification of the test case: prediction is the **majority** class voted for.

5.1 The Nearest Neighbour Algorithm

The **1-nearest neighbour algorithm** is the simplest similarity-based / instance-based method. There is no real training phase, we just store the training cases. Given a query case with a value to be predicted, we compute the distance of the query case from all stored instances and select the nearest neighbour case. We then assign the test case the same label (class or regression value) as its nearest neighbour. The main problem with this approach is susceptibility to noise; to reduce susceptibility to noise, use more than one neighbour, i.e., the k-nearest neighbours algorithm.

1NN with Euclidean distance as the distance metric is equivalent to partitioning the feature space into a **Voronoi Tessellation**: finding the predicted target class is equivalent to finding which Voronoi region it occupies.

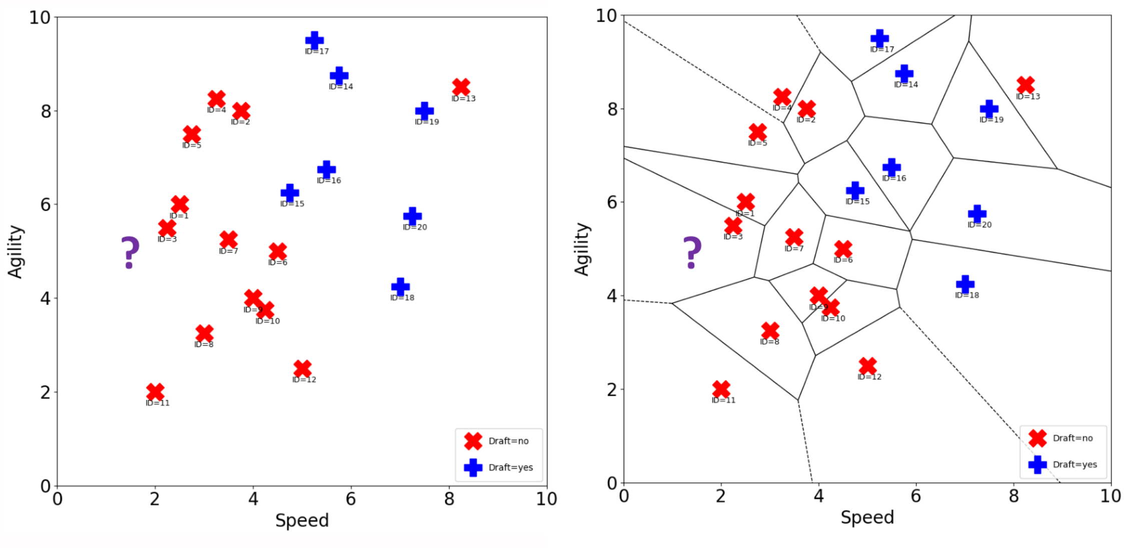


Figure 10: Feature Space Plot (left) & Corresponding Voronoi Tesselation (right)

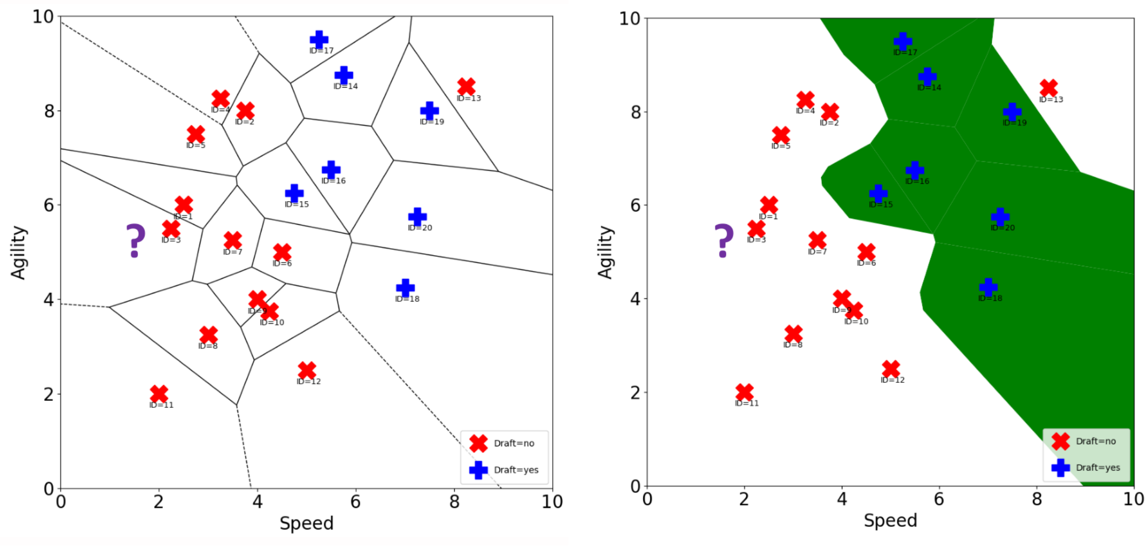


Figure 11: 1NN Decision Boundary from Voronoi Tessellation

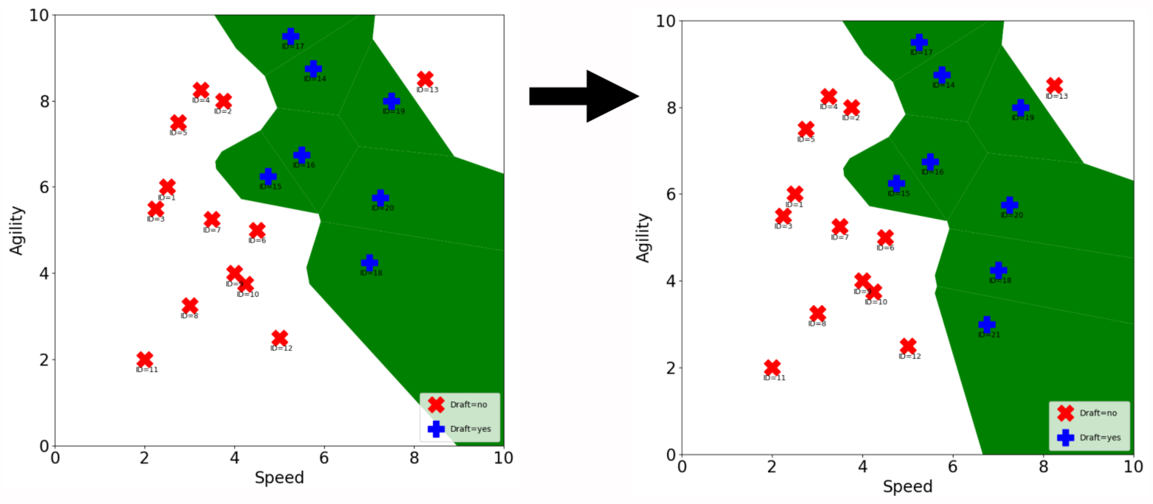


Figure 12: Effect of Adding More Training Data to Voronoi Tessellation

5.2 k-NN Hyperparameters

The k-NN algorithm also introduces a new concept to us that is very important for ML algorithms in general: hyperparameters. In ML algorithms, a **hyperparameter** is a parameter set by the user that is used to control the behaviour of the learning process. Many ML algorithms also have other parameters that are set by the algorithm during its learning process (e.g., the weights assigned to connections between neurons in an artificial neural network). Examples of hyperparameters include:

- Learning rate (typically denoted using the Greek letter α).
- Topology of a neural network (the number & layout of neurons).
- The choice of optimiser when updating the weights of a neural network.

Many ML algorithms are very sensitive to the choice of hyperparameters: poor choice of values yields poor performance. Therefore, hyperparameter tuning (i.e., determining the values that yield the best performance) is an important topic in ML. However, some simple ML algorithms do not have any hyperparameters.

k-NN has several key hyperparameters that we must choose before applying it to a dataset:

- The number of neighbours k to take into account when making a prediction: `n_neighbours` in the scikit-learn implementation of `KNeighboursClassifier`.
- The method used to measure how similar instances are to one another: `metric` in scikit-learn.

5.3 Measuring Similarity

5.3.1 Measuring Similarity Using Distance

Consider the college athletes dataset from earlier. How should we measure the similarity between instances in this case? **Distance** is one option: plot the points in 2D space and draw a straight line between them. We can think of each feature of interest as a dimension in hyperspace.

A **metric** or distance function may be used to define the distance between any pair of elements in a set. metric(a, b) is a function that returns the distance between two instances a & b in a set. a & b are vectors containing the values of the attributes we are interested in for the data points we wish to measure between.

5.3.2 Euclidean Distance

Euclidean distance is one of the best-known distance metrics. It computes the length of a straight line between two points.

$$\text{Euclidean}(a, b) = \sqrt{\sum_{i=1}^{m} (a[i] - b[i])^2}$$

Here m is the number of features / attributes to be used to calculate the distance (i.e., the dimensions of the vectors a & b). Euclidean distance calculates the square root of the sum of squared differences for each feature.

5.3.3 Manhattan Distance

Manhattan distance (also known as "taxicab distance") is the distance between two points measured along axes at right angles.

$$\text{Manhattan}(a, b) = \sum_{i=1}^{m} \text{abs}(a[i] - b[i])$$

As before, m is the number of features / attributes to be used to calculate the distance (i.e., the dimension of the vectors a & b) and abs() is a function which returns the absolute value of a number. Manhattan distance calculates the sum of the absolute differences for each feature.

Example: Calculating Distance

Calculate the distance between $d_{12} = [5.00, 2.50]$ & $d_5 = [2.75, 7.50]$.

$$\text{Euclidean}(d_{12}, d_5) = \sqrt{(5.00 - 2.75)^2 + (2.50 - 7.50)^2} = 5.483$$

$$\text{Manhattan}(d_{12}, d_5) = \text{abs}(5.00 - 2.75) + \text{abs}(2.50 - 7.50) = 7.25$$

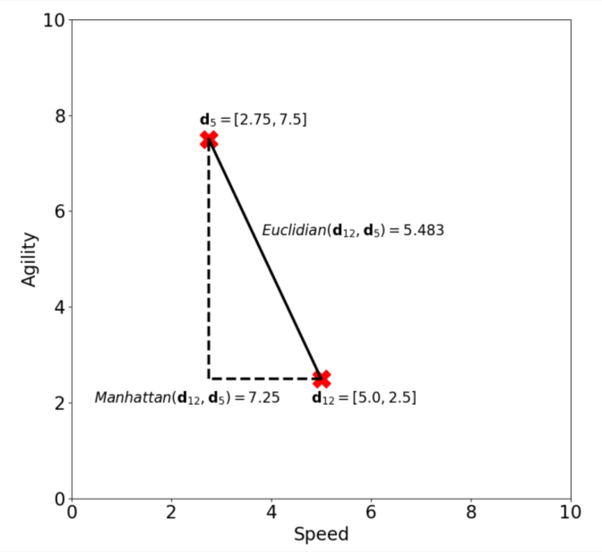


Figure 13: Euclidean vs Manhattan Distance

5.3.4 Minkowski Distance

The **Minkowski distance** metric generalises both the Manhattan distance and the Euclidean distance metrics.

$$\text{Minkowski}(a, b) = \left(\sum_{i=1}^{m} \text{abs}(a[i] - b[i])^p \right)^{\frac{1}{p}}$$

As before, m is the number of features / attributes to be used to calculate the distance (i.e., the dimension of the vectors a & b). Minkowski distance calculates the absolute value of the differences for each feature.

5.3.5 Similarity for Discrete Attributes

Thus far we have considered similarity measures that only apply to continuous attributes[1]. Many datasets have attributes that have a finite number of discrete values (e.g., Yes/No or True/False, survey responses, ratings). One approach to handling discrete attributes is **Hamming distance**: the Hamming distance is calculated as 0 for each attribute where both cases have the same value and 1 for each attribute where they are different. E.g., Hamming distance between the strings "Stephen" and "Stefann" is 3.

5.3.6 Comparison of Distance Metrics

Euclidean & Manhattan distance are the most commonly used distance metrics although it is possible to define infinitely many distance metrics using the Minkowski distance. Manhattan distance is cheaper to compute than Euclidean distance as it is not necessary to compute the squares of differences and a square root, so Manhattan distance may be a better choice for very large datasets if computational resources are limited. It's worthwhile to try out several different distance metrics to see which is the most suitable for the dataset at hand. Many other methods to measure similarity also exist, including cosine similarity, Russel-Rao, Sokal-Michener.

[1] Note that discrete/continuous attributes are not to be confused with classification/regression

5.4 Choosing a Value for k

The appropriate value for k is application dependent, and experimentation is needed to find the optimal value. Typically, it is > 3 and often in the range $5 - 21$. Increasing K has a **smoothing effect**:

- If k is too low, it tends to overfit if the data is noisy.
- If k is too high, it tends to underfit.

In imbalanced datasets, the majority target class tends to dominate for large k values. It's important to note that k does not affect computational cost much: most of the computation is in calculating the distances from the query to all stored instances.

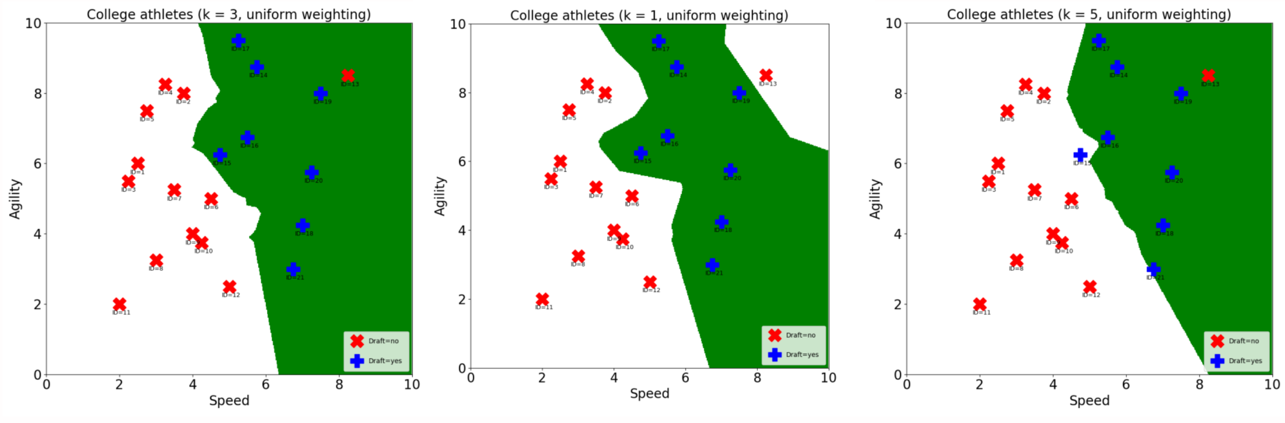


Figure 14: Effect of Increasing k (1)

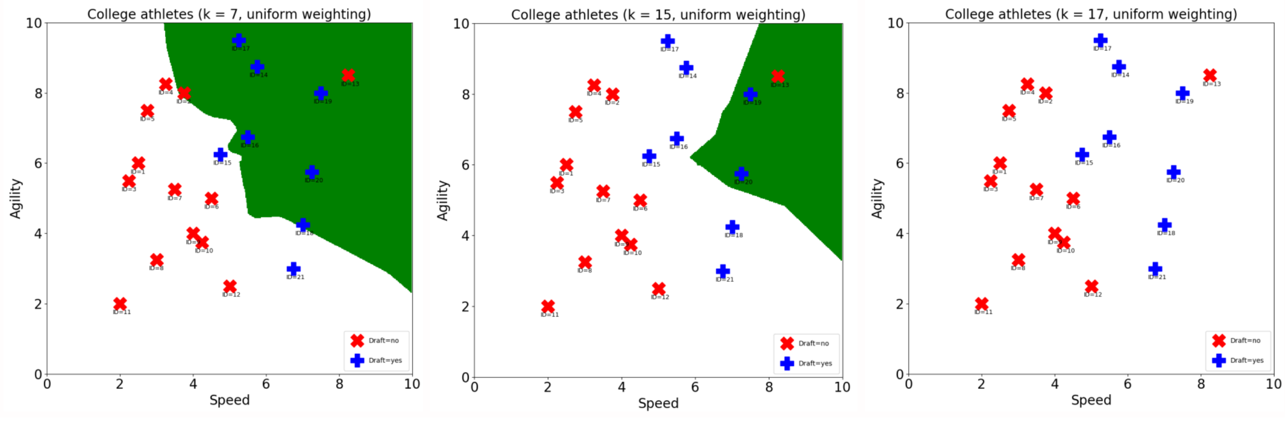


Figure 15: Effect of Increasing k (2)

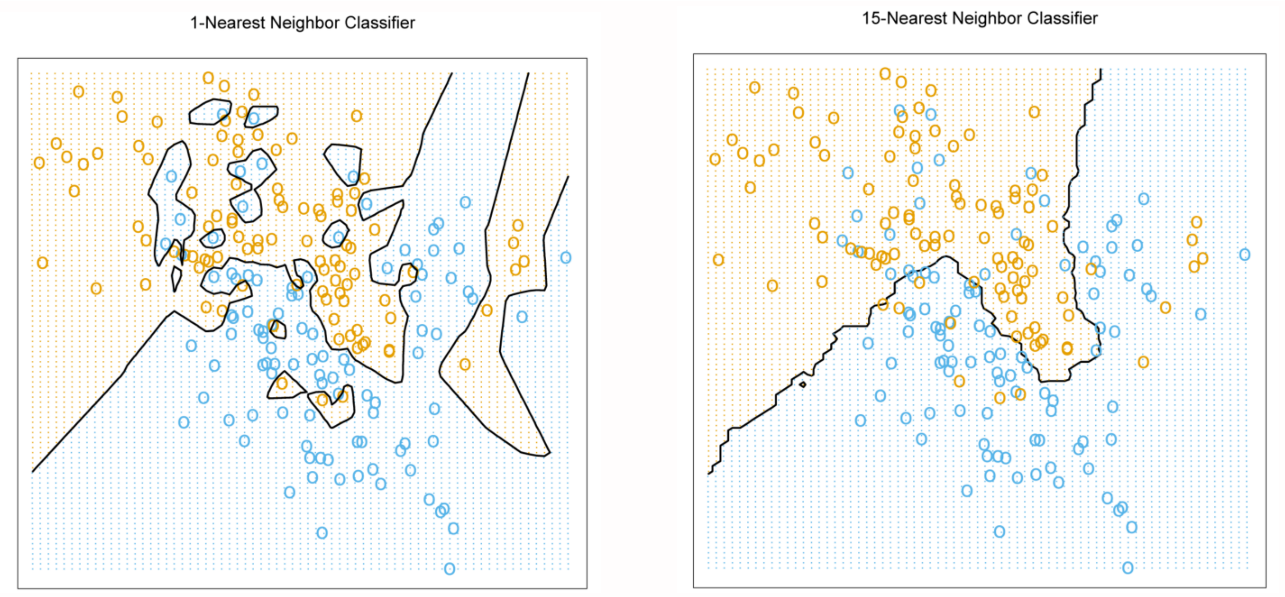


Figure 16: Smoothing Effect of k

5.4.1 Distance-Weighted k-NN

In **distance-weighted** k**-NN**, we give each neighbour a weight equal to the inverse of its distance from the target. We then take the weighted vote or weighted average to classify the target case. It's reasonable to use $k = [\text{all training cases}]$.

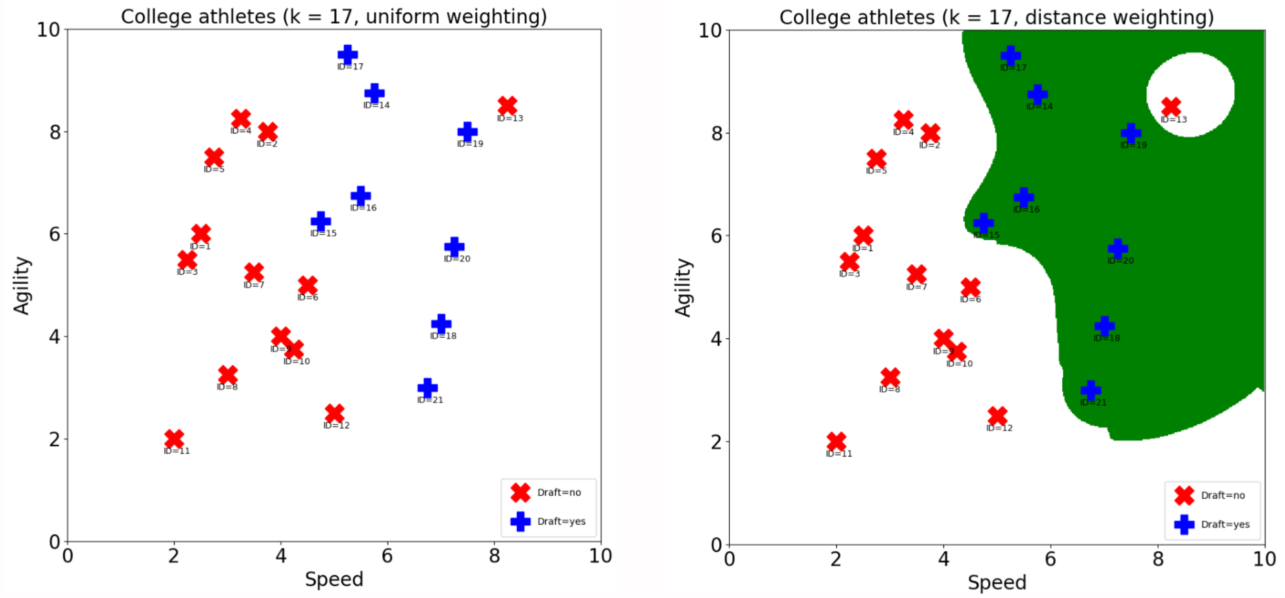


Figure 17: Effect of Distance Weighting

6 Decision Trees

Decision trees are a fundamental structure used in information-based machine learning. The idea is to use a decision tree as a predictive model to decide what category/label/class an item belongs to based on the values of its features. Decision trees consist of **nodes** (where two branches intersect) which are decision points which partition the data. Observations about an item (values of features) are represented using branches. The terminal nodes are called **leaves** and specify the target label for an item. The inductive learning of a decision tree is as follows:

1. For all attributes that have not yet been used in the tree, calculate their impurity (**entropy** or **Gini index**) and **information/Gini gain** values for the training samples.

2. Select the attribute that has the **highest** information gain.
3. Make a tree node containing that attribute.
4. This node **partitions** the data: apply the algorithm recursively to each partition.

The main class used in scikit-learn to implement decision tree learning for classification tasks is `DecisionTreeClassifier`. The default measure of impurity is the Gini index, but entropy is also an option.

6.1 Computing Entropy

We already saw how some descriptive features can more effectively discriminate between (or predict) classes which are present in the dataset. Decision trees partition the data at each node, so it makes sense to use features which have higher discriminatory power "higher up" in a decision tree. Therefore, we need to develop a formal measure of the discriminatory power of a given attribute.

Claude Shannon (often referred to as "the father of information theory") proposed a measure of the impurity of the elements in the set called **entropy**. Entropy may be used to measure the uncertainty of a random variable. The term "entropy" generally refers to disorder or uncertainty, so the use of this term in the context of information theory is analogous to other well-known uses of the term such as in statistical themodynamics. The acquisition of information (**information gain**) corresponds to a **reduction in entropy**.

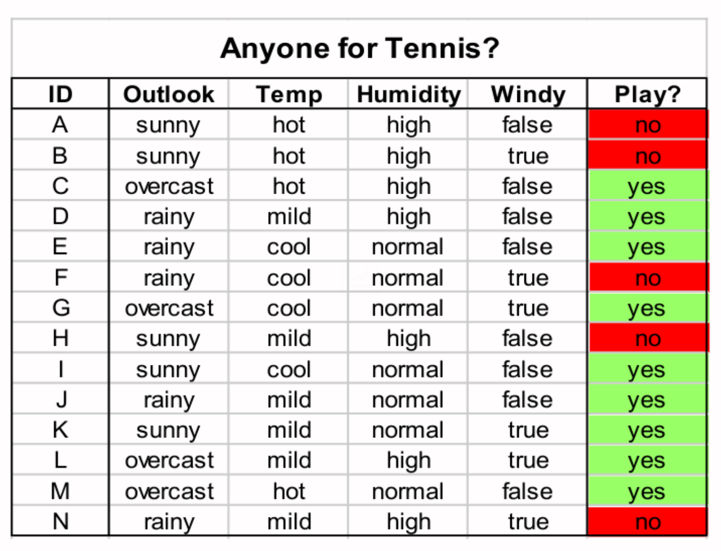
The **entropy** of a dataset S with n different classes may be calculated as:

$$\text{Ent}(S) = \sum_{i=1}^{n} -p_i \log_2 p_i$$

where p_i is the proportion of the class i in the dataset. This is an example of a probability mass function. Entropy is typically measured in **bits** (note the $\log_2$ in the equation above): the lowest possible entropy output from this function is 0 ($\log_2 1 = 0$), while the highest possible entropy is $\log_2 n$ (which is equal to 1 when there are only two classes).

We use the binary logarithm because a useful measure of uncertainty should assign high uncertainty to outcomes with a low probability and assign low uncertainty values to outcomes with a high probability. $\log_2$ returns large negative values when P is close to 0 and small negative values when P is close to 1. We use $-\log_2$ for convenience, as it returns positive entropy values with 0 as the lowest entropy.

Worked Entropy Example



Anyone for Tennis?

ID	Outlook	Temp	Humidity	Windy	Play?
A	sunny	hot	high	false	no
B	sunny	hot	high	true	no
C	overcast	hot	high	false	yes
D	rainy	mild	high	false	yes
E	rainy	cool	normal	false	yes
F	rainy	cool	normal	true	no
G	overcast	cool	normal	true	yes
H	sunny	mild	high	false	no
I	sunny	cool	normal	false	yes
J	rainy	mild	normal	false	yes
K	sunny	mild	normal	true	yes
L	overcast	mild	high	true	yes
M	overcast	hot	normal	false	yes
N	rainy	mild	high	true	no

Figure 18: Example Data

Workings;

$$\begin{aligned}\text{Ent}(S) =& \text{Ent}([9+, 5-]) \\ =& \frac{-9}{14}\log_2\left(\frac{9}{14}\right) - \frac{5}{14}\log_2\left(\frac{5}{14}\right) \\ =& 0.9403\end{aligned}$$

Note that if you are calculating entropy using a spreadsheet application such as Excel, make sure that you are using $\log_2$, e.g. `LOG(9/14,2)`.

6.2 Computing Information Gain

The **information gain** of an attribute is the reduction of entropy from partitioning the data according to that attribute:

$$\text{Gain}(S, A) = \text{Ent}(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|}\text{Ent}(S_v)$$

Here S is the entire set of data being considered and S_v refers to each partition of the data according to each possible value v for the attribute. $|S|$ & $|S_v|$ refer to the cardinality or size of the overall dataset, and the cardinality or size of a partition respectively. When selecting an attribute for a node in a decision tree, we use whichever attribute A that gives the greatest information gain.

Worked Information Gain Example

Given $|S| = 14$, $\left|S_{\text{windy=true}}\right| = 6$, & $\left|S_{\text{windy=false}}\right| = 8$, calculate the information gain of the attribute "windy".

$$\begin{aligned}
\text{Gain}(S, \text{windy}) =& \text{Ent}(S) - \frac{\left|S_{\text{windy=true}}\right|}{|S|}\text{Ent}(S_{\text{windy}} = \text{true}) - \frac{\left|S_{\text{windy=false}}\right|}{|S|}\text{Ent}(S_{\text{windy}} = \text{false}) \\
=& \text{Ent}(S) - \left(\frac{6}{14}\right)\text{Ent}([3+, 3-]) - \left(\frac{8}{14}\right)\text{Ent}([6+, 2-]) \\
=& 0.940 - \left(\frac{6}{14}\right)1.00 - \left(\frac{8}{14}\right)0.811 \\
=& 0.048
\end{aligned}$$

The best partitioning is the one that results in the highest information gain. Once the best split for the root node is found, the procedure is repeated with each subset of examples. S will then refer to the subset in the partition being considered instead of the entire dataset.

6.3 Computing the Gini Index

An alternative to using entropy as the measure of the impurity of a set is to use the **Gini Index**:

$$\text{Gini}(S) = 1 - \sum_{i=1}^{n} p_i^2$$

This is the default measure of impurity in scikit-learn. The gain for a feature can then be calculated based off the reduction in the Gini Index (rather than as a reduction in entropy):

$$\text{GiniGain}(S, A) = \text{Gini}(S) = \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|}\text{Gini}(S_v)$$

6.4 The ID3 Algorithm

Algorithm 1 ID3 Algorithm

```
 1: procedure ID3(Examples, Attributes, Target)
 2:     Input:
 3:         Examples: set of classified examples
 4:         Attributes: set of attributes in the examples
 5:         Target: classification to be predicted
 6:     if Examples is empty then
 7:         return Default class
 8:     else if all Examples have the same class then
 9:         return this class
10:     else if all Attributes are tested then
11:         return majority class
12:     else
13:         Let Best = attribute that best separates Examples relative to Target
14:         Let Tree = new decision tree with Best as root node
15:         for all value $v_i$ of Best do
16:             Let Examples$_i$ = subset of Examples where Best = $v_i$
17:             Let Subtree = ID3(Examples$_i$, Attributes - Best, Target)
18:             Add branch from Tree to Subtree with label $v_i$
19:         end for
20:         return Tree
21:     end if
22: end procedure
```

This code will not be asked for on the exam.

6.5 Decision Tree Summary

Decision trees are popular because:

- It's a relatively easy algorithm to implement.
- It's fast: greedy search without backtracking.
- It has comprehensible output, which is important in decision-making (medical, financial, etc.).
- It's practical.
- It's **expressive:** a decision tree can technically represent any boolean function, although some functions require exponentially large trees such as a parity function.

6.5.1 Dealing with Noisy or Missing Data

If the data is inconsistent or *noisy* we can either use the majority class as in line 11 of the above ID3 algorithm, or interpret the values as probabilities, or return the average target feature value.

For missing data, we could assign the most common value among the training examples that reach that node, or we could assume that the attribute has all possible values, weighting each value according to its frequency among the training examples that reach that node.

6.5.2 Instability of Decision Trees

The hypothesis found by decision trees is sensitive to the training set used as a consequence of the greedy search used. Some ideas to reduce the instability of decision trees include altering the attribute selection procedure, so that the tree learning algorithm is less sensitive to some percentage of the training dataset being replaced.

6.5.3 Pruning

Overfitting occurs in a predictive model when the hypothesis learned makes predictions which are based on spurious patterns in the training data set. The consequence of this is poor generalisation to new examples. Overfitting may happen for a number of reasons, including sampling variance or noise present in the dataset. **Tree pruning** may be used to combat overfitting in decision trees. However, tree pruning can also lead to induced trees which are inconsistent with the training set.

Generally, there are two different approaches to pruning:

- Pre-pruning
- Post-pruning

7 Model Evaluation

The most important part of the design of an evaluation experiment for a predictive model is not the same as the data used to train the model. The purpose of evaluation is threefold:

- To determine which model is the most suitable for a task.
- To estimate how the model will perform.
- To demonstrate to users that the model will meet their needs.

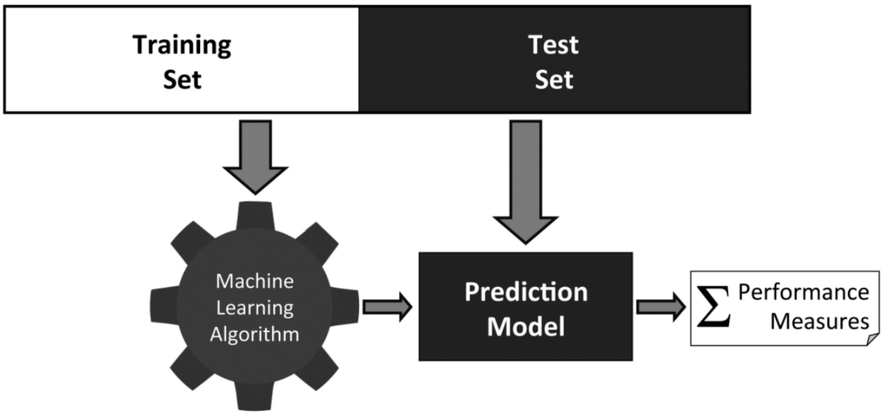
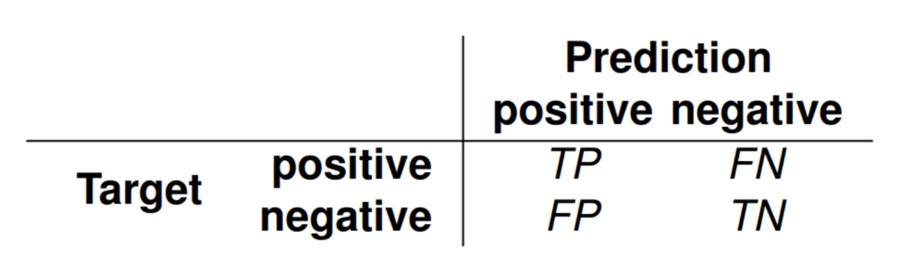


Figure 19: Using a Hold-Out Test Set

By convention, in binary classification tasks we refer to one class as *positive* and the other class as *negative*. There are four possible outcomes for a binary classification task:

- TP: True Positive.
- TN: True Negative.
- FP: False Positive.
- FN: False Negative.

These are often represented using a **confusion matrix**.



		Prediction	
		positive	negative
Target	positive	*TP*	*FN*
	negative	*FP*	*TN*

Figure 20: Using a Hold-Out Test Set

The **misclassification rate** can be calculated as:

$$\begin{aligned}\text{misclassification rate} &= \frac{\text{\# of incorrect predictions}}{\text{total predictions}}\\ &= \frac{(FP+FN)}{(TP+TN+FP+FN)}\end{aligned}$$

The classification accuracy can be calculated as:

$$\text{classification accuracy} = \frac{(TP+TN)}{(TP+TN+FP+FN)}$$

7.1 Metrics for Binary Classification Tasks

Confusion matrix-based measures for binary categorical targets include:

- TPR: True Positive Rate.

$$TPR = \frac{TP}{(TP+FN)}$$

- TNR: True Negative Rate.

$$TNR = \frac{TN}{(TN+FP)}$$

- FPR: False Positive Rate.

$$FPR = \frac{FP}{(TN+FP)}$$

- FNR: False Negative Rate.

$$FNR = \frac{FN}{(TP+FN)}$$

All these measures can have values in the range of 0 to 1. Higher values of TPR & TNR, and lower values of FNR & FPR, indicate better model perfomance.

7.1.1 Precision & Recall

Precision captures how often, when a model makes a positive prediction, that prediction turns out to be correct.

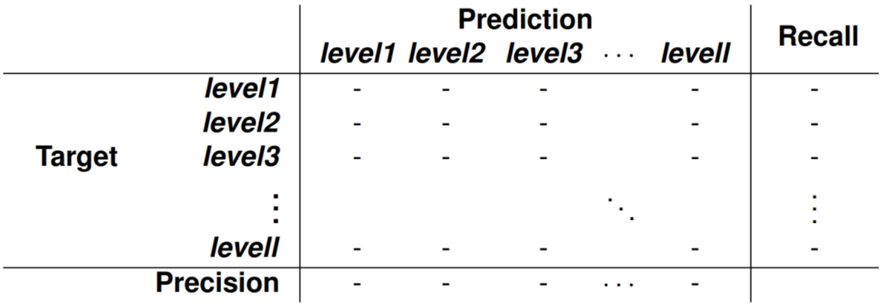
$$\text{precision} = \frac{TP}{(TP+FP)}$$

Recall is equivalent to the true positive rate, and tells us how confident we can be that all the instances with the positive target level have been found by the model. Both precision & recall have values in the range 0 to 1.

$$\text{precision} = \frac{TP}{(TP+FN)}$$

7.2 Multinomial Classification Tasks

All the measures we have discussed hitherto apply to binary classification problems only. Many classification problems are multinomial, i.e., have more that two target levels / classes. We can easily extend the confusion matrix concept to multiple target levels by adding a row & a column for each level.



		Prediction					Recall
		level1	*level2*	*level3*	···	*levelI*	
	level1	-	-	-		-	-
	level2	-	-	-		-	-
Target	*level3*	-	-	-		-	-
	⋮				⋱		⋮
	levelI	-	-	-		-	-
	Precision	-	-	-	···	-	

Figure 21: Extended Confusion Matrix

We can also calculate precision & recall for each target level independently. $TP(\mathrm{I})$ refers to the number of instances correctly assigned a prediction of class I. $FP(\mathrm{I})$ refers to the number of instances incorrectly assigned a prediction of class I. $FN(\mathrm{I})$ refers to the number of instances that should've been assigned a prediction of class I but were given some other prediction.

$$\text{precision}(\mathrm{I}) = \frac{TP(\mathrm{I})}{TP(\mathrm{I}) + FP(\mathrm{I})}$$

$$\text{recall}(\mathrm{I}) = \frac{TP(\mathrm{I})}{TP(\mathrm{I}) + FN(\mathrm{I})}$$

Confusion matrices can be easily created in scikit-learn using the built-in classes.

7.3 Cross-Validation & Grid Search

So far, we have just manually divided data into training & test sets. We want to avoid problems with a "lucky split" where most difficult examples end up in the training set and most easy examples end up in the test set. Methods like k-fold cross-validation allow us to use all examples for both training & testing.

7.3.1 k-Fold Cross-Validation

k-fold cross validation (CV) allows all of the data to be used for both training & testing. The procedure is as follows:

1. Split the data into k different folds.
2. Use $k - 1$ folds for training, and the remaining fold for testing.
3. Repeat the entire process k times, using a different fold for testing each time. Report the per-fold accuracy & the average accuracy for both training & testing.

Cross-validation can be easily implemented in scikit-learn by calling the `sklearn.model_selection.cross_val_score()` helper method on the estimator & the dataset. This will return the testing accuracy only. If it is desired to report the training accuracy or other metrics, scikit-learn provides the method `sklearn.mode_selection.cross_validate` which provides additional options. Training scores will be returned in the parameter `return_train_score` of this method is set to **`True`**. The score parameter can be used to specify the metrics that will be computed to score the models trained during cross-validation.

7.3.2 Hyperparameter Optimisation

scikit-learn provides a class `sklearn.model_selection.GridSearchCV` that allows an exhaustive search through ranges of specified hyperparameter values. Scores are calculated for each possible hyperparameter combination on the grid, allowing the combination with the best score to be identified. It is widely used when performing hyperparameter tuning for ML models.

7.4 Prediction Scores & ROC Curves

Many different classification algorithms produce **prediction scores**, e.g. Näive Bayes, logistic regression, etc. The score indicates the system's certainty that the given observation belongs to the positive class. To make the decision about whether the observation should be classified as positive or negative as a consumer of this score, one will interpret the score by picking a classification threshold (cut-off) and comparing the score against it. Any observations with scores higher than the threshold are then predicted as the positive class and scores lower than the threshold are predicted as the negative class. Often, a prediction score greater than or equal to 0.5 is interpreted as the positive class by default, and a prediction with a score less than 0.5 is interpreted as the negative class. The prediction score threshold can be changed, leading to different performance metric values and a different confusion matrix.

As the threshold increases, TPR decreases, and TNR increases. As the threshold decreases, TPR increases, and TNR decreases. Capturing this trade-off is the basis of the **Receiver Operating Characteristic (ROC)** curve.

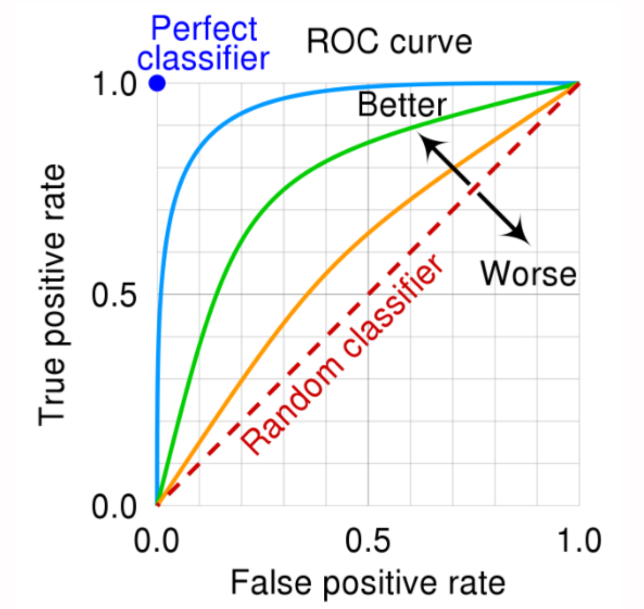


Figure 22: ROC Curve

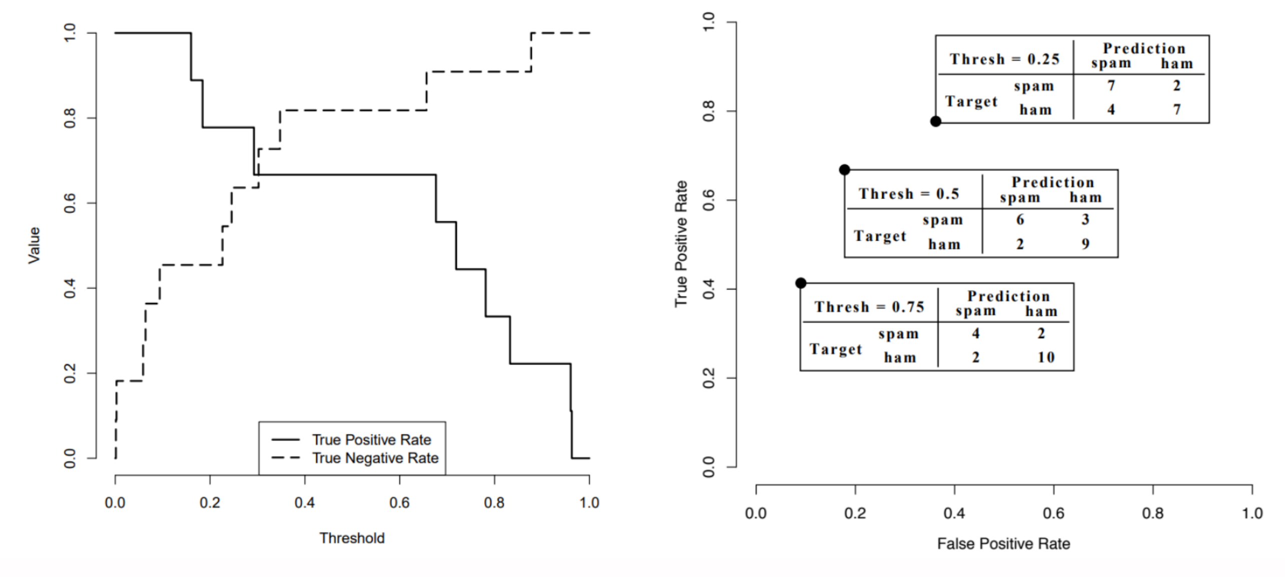


Figure 23: Effect of Increasing the Classification Threshold

7.4.1 ROC Curves

An ideal classifier would be in the top left corner, i.e. TPR = 1.0 and FPR = 0.0. The area under the ROC curve is often used to determine the "strength" of a classifier: greater area indicates a better classifier, and an ideal classifier has an area of 1.0.

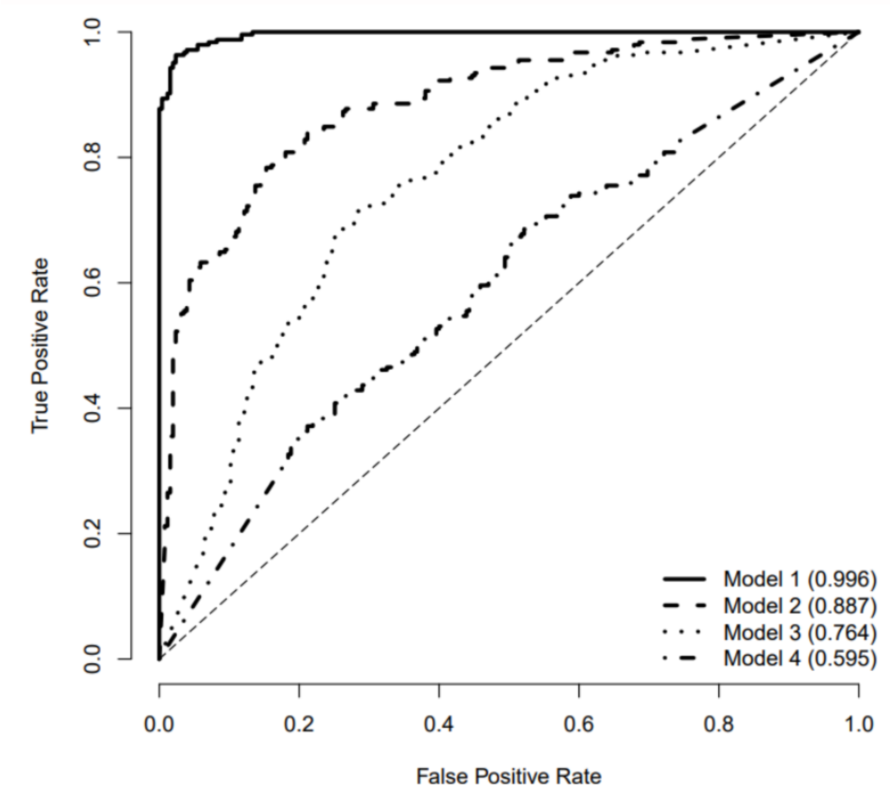


Figure 24: Sample ROC Curves for 5 Different Models Trained on the Same Dataset

scikit-learn provides various classes to generate ROC curves: `https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html`. The link also gives some details on how ROC curves can be computed for mutli-class classification problems.

8 Data Processing & Normalisation

8.1 Data Normalisation

One major problem in data normalisation is **scaling**. For example, if Attribute 1 has a range of 0-10 and Attribute 2 has a range from 0-1000, then Attribute 2 will dominate calculations. The solution for this problem is to rescale all dimensions independently:

- Z-normalisation: calculated by subtracting the population mean from an individual raw score and then dividing it by the population standard deviation.

  $$z = \frac{x - \mu}{\sigma}$$

  where z is the z-score, x is the raw score, μ is the population mean, & σ is the standard deviation of the population.

  This can be achieved in scikit-learn using the `StandardScaler` utility class.

- Min-Max data scaling: also called 0-1 normalisation or range normalisation.

  $$X' = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}}$$

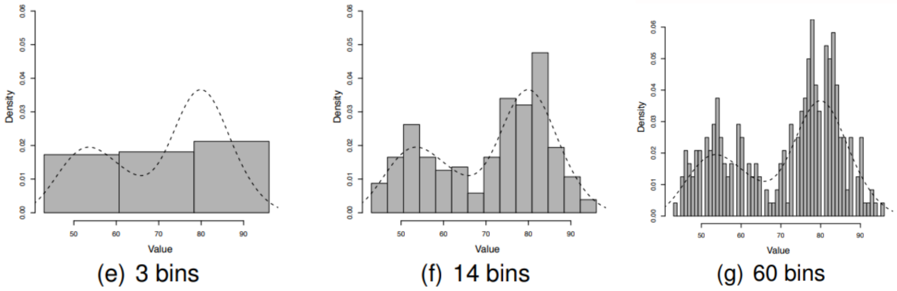
  This can be achieved in scikit-learn using the `MinMaxScaler` utility class.

It is generally good practice to normalise continuous variables before developing an ML model. Some algorithms (e.g., k-NN) are much more susceptible to the effects of the relative scale of attributes than others (e.g., decision trees are more robust to the effects of scale).

8.2 Binning

Binning involves converting a continuous feature into a categorical feature. To perform binning, we define a series of ranges called **bins** for the continuous feature that corresponds to the levels of the new categorical feature we are creating. Two of the more popular ways of defining bins include equal-width binning & equal-frequency binning.

Deciding on the number of bins can be complex: in general, if we set the number of bins to a very low number we may lose a lot of information, but if we set the number of bins to a very high number then we might have very few instances in each bin or even end up with empty bins.

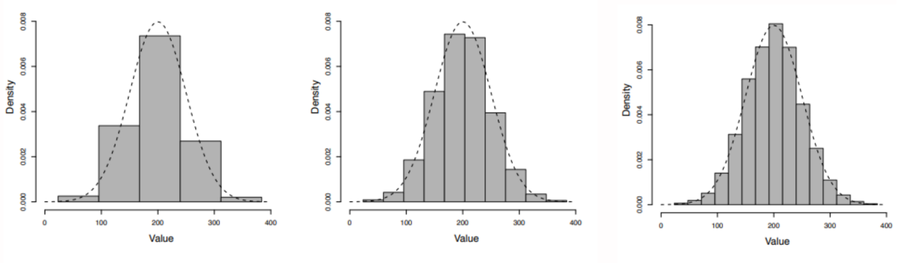


(e) 3 bins (f) 14 bins (g) 60 bins

Figure 25: The effect of different numbers of bins

8.2.1 Equal-Width Binning

The **equal-width binning** approach splits the range of the feature values into b bins, each of size $\frac{\text{range}}{b}$.

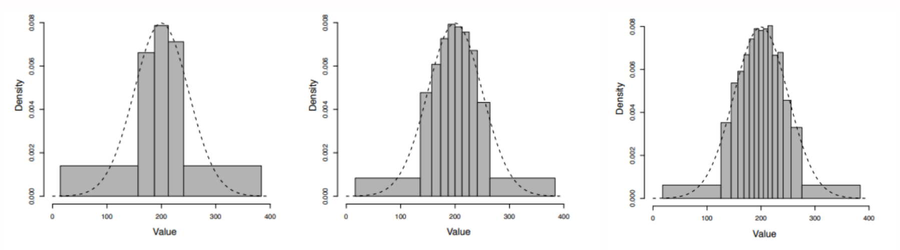


(h) 5 Equal-width bins (i) 10 Equal-width bins (j) 15 Equal-width bins

Figure 26: Equal-width binning

8.2.2 Equal-Frequency Binning

Equal-frequency binning first sorts the continuous feature values into ascending order, and then places an equal number of instances into each bin, starting with bin 1. The number of instances placed in each bin is simply the total number of instances divided by the number of bins b.



(k) 5 Equal-frequency bins (l) 10 Equal-frequency bins (m) 15 Equal-frequency bins

Figure 27: Equal-frequency binning

8.3 Sampling

Sometimes, the dataset that we have is so large that we do not use all the data available to us and instead take a smaller percentage from the larger dataset. For example, we may wish to only use part of the data because training will take a long time with very many examples for some algorithms. In the case of k-NN, a very large training set may lead to long prediction times. However, we need to be careful when sampling to ensure that the resulting datasets are still representative of the original data and that no unintended bias is introduced during this process. Common forms of sampling include: top sampling, random sampling, stratified sampling, under-sampling, & over-sampling.

8.3.1 Top Sampling

Top sampling simply selects the top s% of instances from a dataset to create a sample. It runs a serious risk of introducing bias as the sample will be affected by any ordering of the original dataset; therefore, top sampling should be avoided.

8.3.2 Random Sampling

Random sampling is a good default sampling strategy as it randomly selects a proportion (s%) of the instances from a large dataset to create a smaller set. It is a good choice in most cases as the random nature of the selection of instances should avoid introducing bias.

8.3.3 Stratified Sampling

Stratified sampling is a sampling method that ensures that the relative frequencies of the levels of a specific stratification feature are maintained in the sampled dataset. To perform stratified sampling, the instances in a dataset are divided into groups (or *strata*), where each group contains only instances that have a particular level for the stratification feature. s% of the instances in each stratum are randomly selected and these selections are combined to give an overall sample of s% of the original dataset.

In contrast to stratified sampling, sometimes we would like a sample to contain different relative frequencies of the levels of a particular discrete feature to the distribution in the original dataset. To do this, we can use under-sampling or over-sampling.

8.3.4 Under-Sampling

Under-sampling begins by dividing a dataset into groups, where each group contains only instances that have a particular level for the feature to be under-sampled. The number of instances in the smallest group is the under-sampling target size. Each group containing more instances than the smallest one is then randomly sampled by the appropriate percentage to create a subset that is the under-sampling target size. These under-sampled groups are then combined to create the overall under-sampled dataset.

8.3.5 Over-Sampling

Over-sampling addresses the same issue as under-sampling but in the opposite way: after dividing the dataset into groups, the number of instances in the largest group becomes the over-sampling target size. From each smaller group, we then create a sample containing that number of instances using random sampling without replacement. These larger samples are combined to form the overall over-sampled dataset.

8.4 Feature Selection

8.4.1 The Curse of Dimensionality

Some attributes are much more significant than others are considered equally in distance metric, possibly leading to bad predictions. k-NN uses all attributes when making a prediction, whereas other algorithms (e.g., decision trees) use only the most useful features and so are not as badly affected by the curse of dimensionality. Any algorithm that considers all attributes in a high-dimensional space equally has this problem, not just k-NN + Euclidean distance. Two solutions to the curse of dimensionality are:

- Assign weighting to each dimension (not the same as distance-weighted k-NN). Optimise weighting to minimise error.

- Give some dimensions 0 weight: feature subset selection.

Consider cases with d dimensions, in a hypercube of *unit* volume. Assume that neighbourhoods are hypercubes with length b; volume is b^d. To contain k points, the average neighbourhood must occupy $\frac{k}{N}$ of the entire volume.

$$
\begin{aligned}
&\Rightarrow b^d = \frac{k}{N} \\
&\Rightarrow b = \left(\frac{k}{N}\right)^{\frac{1}{d}}
\end{aligned}
$$

In high dimensions, e.g. $k = 10$, $N = 1,000,000$, $d = 100$, then $b = 0.89$, i.e., a neighbourhood must occupy 90% of each dimension of space. In low dimensions, with $k = 10$, $N = 1,000,000$, $d = 2$, then $b = 0.003$ which is acceptable. High dimensional spaces are generally very sparse, and each neighbour is very far away.

8.4.2 Feature Selection

Fortunately, some algorithms partially mitigate the effects of the curse of dimensionality (e.g., decision tree learning). However, this is not true for all algorithms, and heuristics for search can sometimes be misleading. k-NN and many other algorithms use all attributes when making a prediction. Acquiring more data is not always a realistic option; the best way to avoid the curse of dimensionality is to use only the most useful features during learning: **feature selection**.

We may wish to distinguish between different types of descriptive features:

- **Predictive:** provides information that is useful when estimating the correct target value.

- **Interacting:** provides useful information only when considered in conjunction with other features.
- **Redundant:** features that have a strong correlation with another feature.
- **Irrelevant:** doesn't provide any useful information for estimating the target value.

Ideally, a good feature selection approach should identify the smallest subset of features that maintain prediction performance.

8.4.3 Feature Selection Approaches

- **Rank & Prune:** rank features according to their predictive power and keep only the top X%. A **filter** is a measure of the predictive power used during ranking, e.g., information gain. A drawback of rank & prune is that features are evaluated in isolation, so we will miss useful *interacting features*.
- **Search for useful feature subsets:** we can pick out useful interacting features by evaluating feature subsets. We could generate, evaluate, & rank all possible feature subsets then pick the best (essentially a brute force approach, computationally expensive). A better approach is a **greedy local search**, which builds the feature subset iteratively by starting out with an empty selection, then trying to add additional features incrementally. This requires evaluation experiments along the way. We stop trying to add more features to the selection once termination conditions are met.

8.5 Covariance & Correlation

As well as visually inspecting scatter plots, we can calculate formal measures of the relationship between two continuous features using **covariance** & **correlation**.

8.5.1 Measuring Covariance

For two features a & b in a dataset of n instances, the **sample covariance** between a & b is:

$$\text{cov}(a, b) = \frac{1}{n-1} \sum_{i=1}^{n} \left((a_i - \overline{a}) \times \left(b_i - \overline{b} \right) \right)$$

where a_i & b_i are the i^{th} instances of features a & b in a dataset, and $\overline{a}$ & $\overline{b}$ are the sample means of features a & b.

Covariance values fall into the range $[-\infty, \infty]$, where negative values indicate a negative relationship, positive values indicate a positive relationship, & values near to zero indicate that there is little to no relationship between the features.

Covariance is measured in the same units as the features that it measures, so comparing something like weight of a basketball player to the height of a basketball player doesn't really make sense as the features are not in the same units. To solve this problem, we use the **correlation coefficient**, also known as the Pearson product-moment correlation coefficient or Pearson's r.

8.5.2 Measuring Correlation

Correlation is a normalised form of covariance with range $[-1, 1]$. The correlation between two features a & b can be calculated as

$$\text{corr}(a, b) = \frac{\text{cov}(a, b)}{\text{sd}(a) \times \text{sd}(b)}$$

where $\text{cov}(a, b)$ is the covariance between features a & b, and $\text{sd}(a)$ & $\text{sd}(b)$ are the standard deviations of a & b respectively.

Correlation values fall into the range $[-1, 1]$ where values close to -1 indicate a very strong negative correlation (or covariance), values close to 1 indicate a very strong positive correlation, & values around 0 indicate no correlation. Features that have no correlation are said to be **independent**.

The **covariance matrix**, usually denoted as $\sum$, between a set of continuous features $\{a, b, \ldots, z\}$, is given as

$$\sum_{\{a,b,\ldots,z\}} = \begin{bmatrix} \text{var}(a) & \text{cov}(a,b) & \cdots & \text{cov}(a,z) \\ \text{cov}(a,b) & \text{var}(b) & \cdots & \text{cov}(b,z) \\ \vdots & \vdots & \vdots & \vdots \\ \text{cov}(z,a) & \text{cov}(z,b) & \cdots & \text{var}(z) \end{bmatrix}$$

Similarly, the **correlation matrix** is just a normalised version of the covariance matrix and shows the correlation between each pair of features:

$$\underset{\{a,b,\ldots,z\}}{\text{correlation matrix}} = \begin{bmatrix} \text{corr}(a,a) & \text{corr}(a,b) & \cdots & \text{corr}(a,z) \\ \text{corr}(b,a) & \text{corr}(b,b) & \cdots & \text{corr}(b,z) \\ \vdots & \vdots & \cdots & \vdots \\ \text{corr}(z,a) & \text{corr}(z,b) & \cdots & \text{corr}(z,z) \end{bmatrix}$$

Correlation is a good measure of the relationship between two continuous features, but is not perfect. Firstly, the correlation measure given earlier responds only to linear relationships between features. In a linear relationship between two features, as one feature increases or decreases, the other feature increases or decreases by a corresponding amount. Frequently, features will have a very strong non-linear relationship that correlation does not respond to. Some limitations of measuring correlation are illustrated very clearly in the famous example of Anscombe's Quartet, published by the famous statistician Francis Anscombe in 1973.

9 Regression

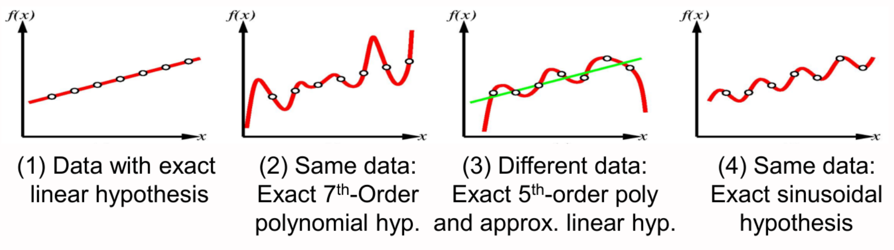
Heretofore, we have looked primarily at classification supervised learning tasks, where the goal is to predict one class from a finite number of possible discrete classes. In **regression** tasks, we also have labelled training & testing data, but the labels take the form of floating-point values (real numbers): the goal is to predict a floating-point number, not a class. Examples of algorithms for regression tasks include:

- Linear regression.
- Decision trees.
- k-nearest neighbours.
- Neural networks.

9.1 Supervised Learning Considerations

9.1.1 Inconsistent Hypotheses

Various hypotheses can be consistent with observations but inconsistent for each other: which one should we choose?



(1) Data with exact linear hypothesis

(2) Same data: Exact 7th-Order polynomial hyp.

(3) Different data: Exact 5th-order poly and approx. linear hyp.

(4) Same data: Exact sinusoidal hypothesis

Figure 28: Hypotheses consistent with the observations but inconsistent with each other

One solution to this problem is **Ockham's Razor**: prefer the simplest hypothesis consistent with the data. However, definitions of simplicity (and consistency) may be subject to debate and it depends strongly on how hypotheses are expressed.

Another consideration to be made is whether or not the hypothesis language is too limited: we might be unable to find a hypothesis that exactly matches the true function. If the true function is more complex than what the hypothesis can express, it will **undefit** the data. If the hypothesis language cannot exactly match the true function, there will be a trade-offf between the complexity of the hypothesis and how well it fits the data. If the hypothesis language is very expressive, its search space will be very large and the computational complexity of finding a good hypothesis will be high. We will also need a large amount of data to avoid **overfitting**.

We can't forget that *we never know the true underlying function*. For example, to avoid the problem with poorly fitting data, we could change the algorithm so that, as well as a searching for the coefficients of polynomials, it tries combinations of trigonometric functions: the learning problem will become enormously more complex but it will probably not solve our problems as we could easily think up some different kind of mathematical function to generate a new dataset that the algorithm still cannot represent perfectly. For this reason, we often use relatively simple hypothesis languages in the absence of special knowledge about the domain: more complex languages don't come with any real guarantees, and more simple languages correspond to easier searching.

9.1.2 Noise, Overfitting, & Underfitting

Noise consists of imprecise or incorrect attribute values or labels. We can't always quantify it, but we should know from the situation if it is present. For example, labels may require subjective judgements or values may come from imprecise judgements.

If the data might have noise, it is harder to decide which hypothesis is best. If you increase the complexity of the hypothesis, you increase ability to fit the data but might also increase the risk of overfitting.

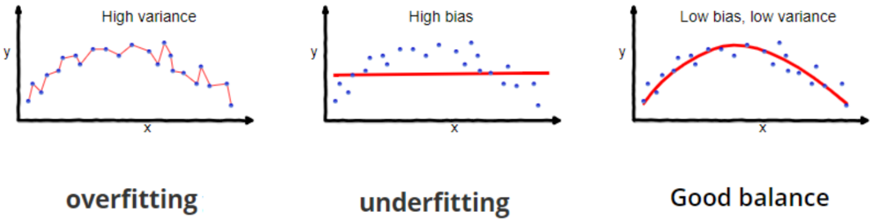
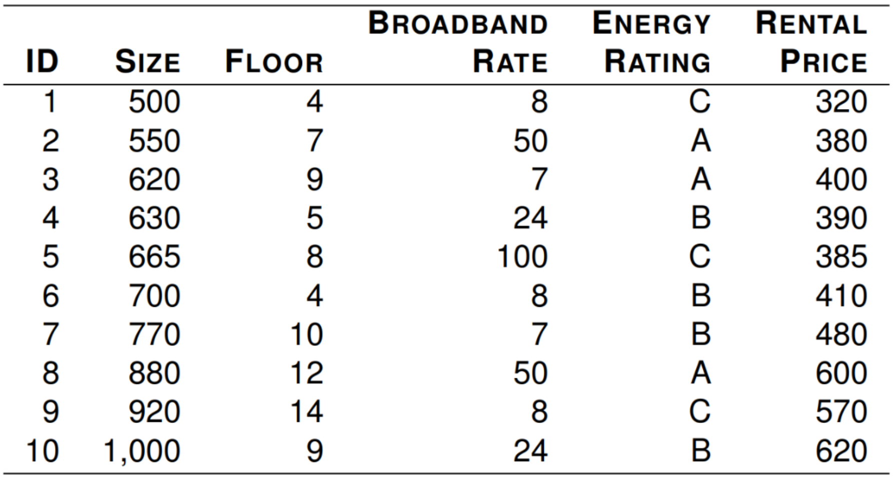


Figure 29: Bias & variance

9.2 Linear Regression Models

We will be referring to the following office rentals example regression dataset throughout this section:



ID	Size	Floor	Broadband Rate	Energy Rating	Rental Price
1	500	4	8	C	320
2	550	7	50	A	380
3	620	9	7	A	400
4	630	5	24	B	390
5	665	8	100	C	385
6	700	4	8	B	410
7	770	10	7	B	480
8	880	12	50	A	600
9	920	14	8	C	570
10	1,000	9	24	B	620

Figure 30: A dataset that includes office rental prices & a number of descriptive features for 10 Dublin city-centre offices

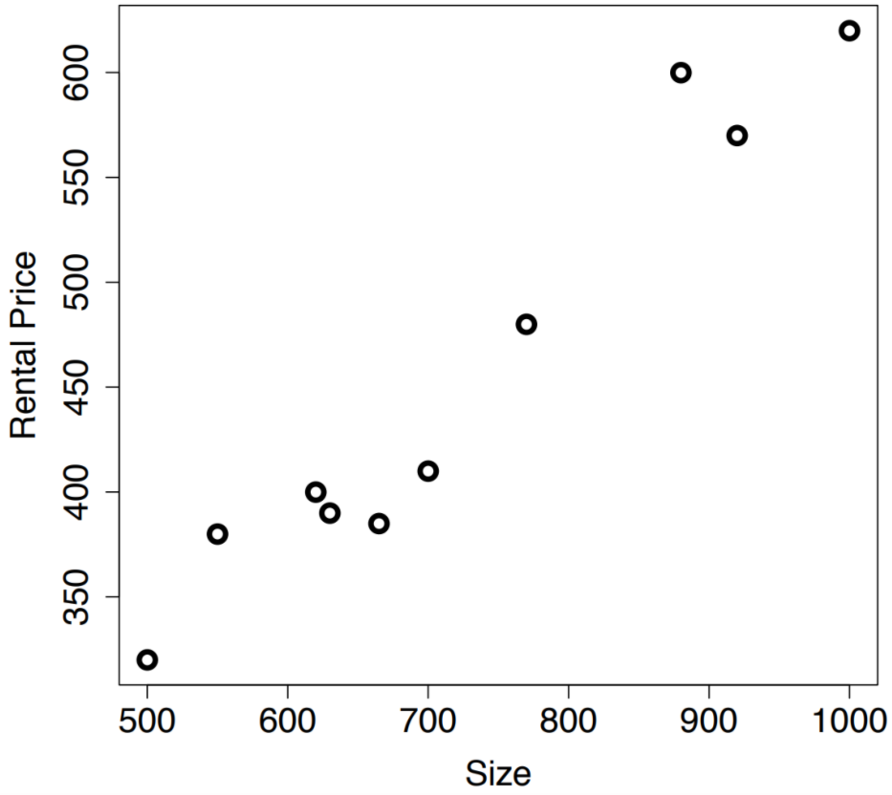


Figure 31: Scatter plot of size vs rental price

9.2.1 Parameterised Prediction Models

A **parameterised prediction model** is initialised with a set of random parameters and an error function is used to judge how well this initial model performs when making predictions for instances in a training dataset. Based on the value of the error function, the parameters are iteratively adjusted to create a more & more accurate model. This is the approach taken by many common ML models, e.g. simple linear regression & neural networks.

9.2.2 Developing a Simple Linear Regression Model

From the scatter plot in Figure 31, it appears that there is a linear relationship between the size and the rental price. The equation of a line can be written as $y = mx + c$. The below scatter plot shows the same scatter plot as in Figure 31, but with a simple linear model added to capture the relationship between office sizes & rental prices. This model is rental price $= 6.47 + 0.62 \times$ size. We can use this model to determine the expected rental price of 730 square foot office: rental price $= 6.47 + 0.62 \times 730 = 459$.

Multivariate linear regression using vector notation can be represented as:

$$\begin{aligned}\mathbb{M}(d) =& w[0] \times d[0] + w[1] \times d[1] + \cdots + w[m] \times d[m] \\ =& \sum_{j=0}^{m} w[j] \times d[j] \\ =& w \cdot d\end{aligned}$$

where:

- w is a vector of model weights,
- m is the number of independent variables,
- $\mathbb{M}_w(d)$ is the predicted value,

- $w \cdot d$ is the vector dot product.

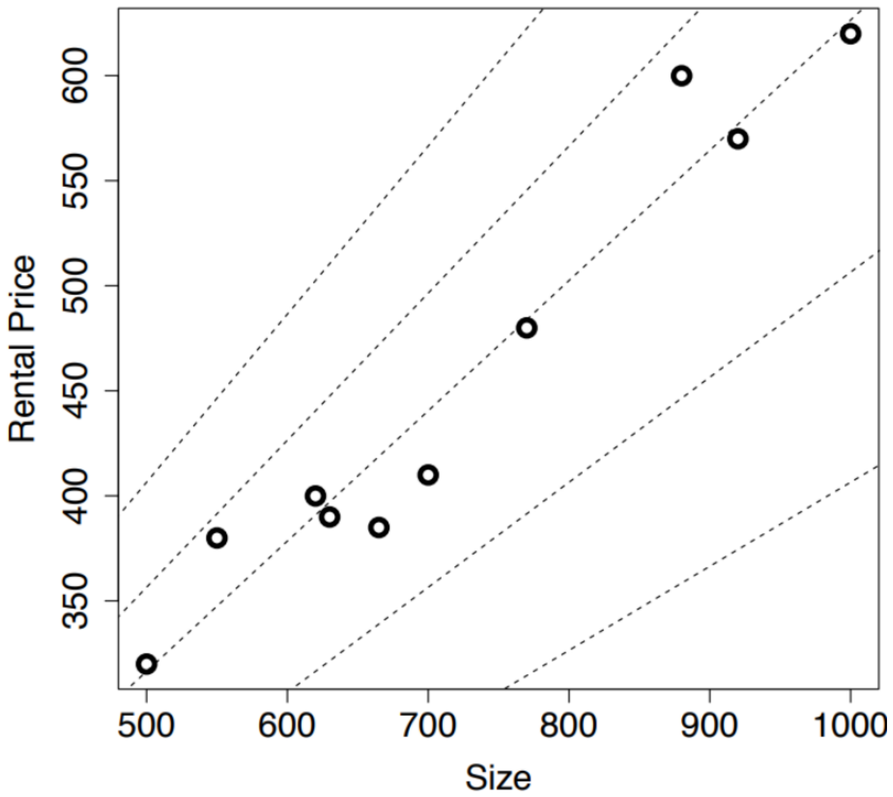


Figure 32: Scatter plot of the SIZE & RENTAL PRICE features For all models in the above scatter plot, $w[0]$ is set to 6.47. From top to bottom, the models use 0.4, 0.5, 0.62, 0.7. & 0.8 respectively for $w[1]$.

For linear regression in one independent variable, we have only two components in the weight vector: $w[0]$ (intercept) & $w[1]$ (slope), and we make predictions based on the value of one independent variable (SIZE in this case).

9.2.3 Measuring Error

Error is measured between the predicted value and the target value. Note that errors may be positive or negative, i.e., above or below the regression line.

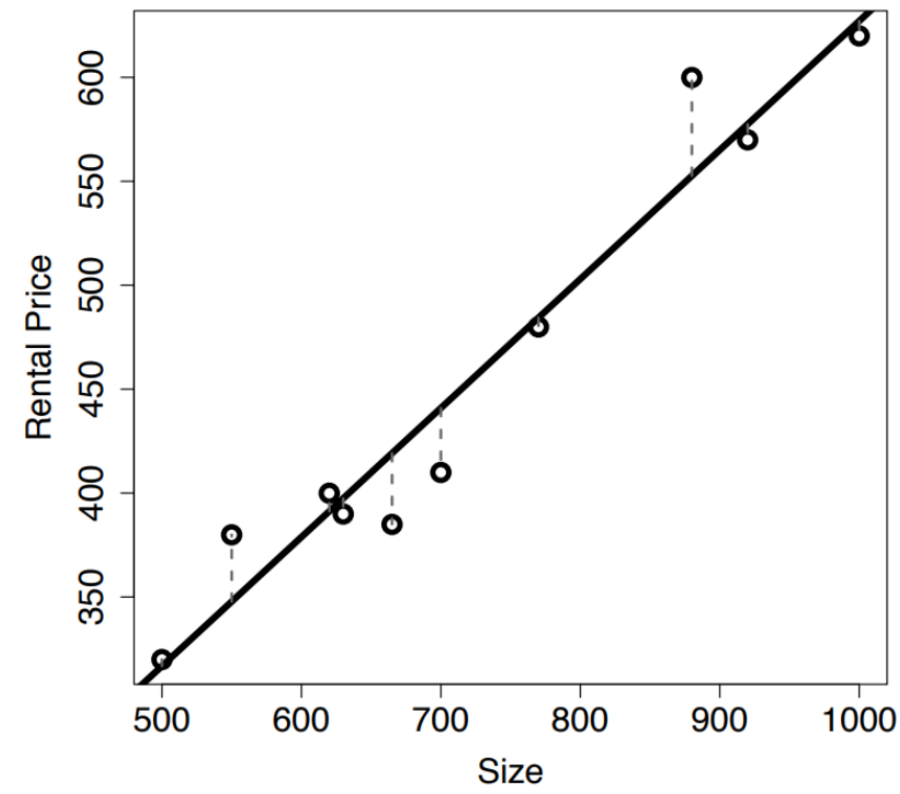


Figure 33: A scatter plot of the SIZE & RENTAL PRICE features from the office rentals dataset showing a candidate prediction model (with $w[0] = 6.47$ & $w[1] = 0.62$) and the resulting errors.

$$\text{sum of squared errors} = \frac{1}{2}\sum_{i=1}^{n}\left(t_i - \mathbb{M}\left(d_i\right)\right)^2$$

In order to formally measure the fit of a linear regression model with a set of training data, we require an error function that captures the error between the predictions made by a model & the actual values in a training dataset. Here $t_i \dots t_n$ is the set of n target values and $\mathbb{M}(d_1) \dots \mathbb{M}(d_n)$ is the set of predictions. By minimising the sum of squared errors or L_2, we can develop a best fit linear regression model. Note that some errors are positive and some errors are negative; if we simply add these errors together, the positive and negative errors would cancel each other out. Therefore, we use the sum of the squared errors rather than the sum of errors because this means that all values will be positive.

The x-y plane is known as the **weight space** and the surface is known as the **error surface**. The model that best fits the training data is the model corresponding to the lowest point on the error surface. One approach to find this point is the **gradient descent algorithm** but we will not cover it in this module. The same concepts apply to multivariate linear regression, although error surfaces cannot easily be visualised.

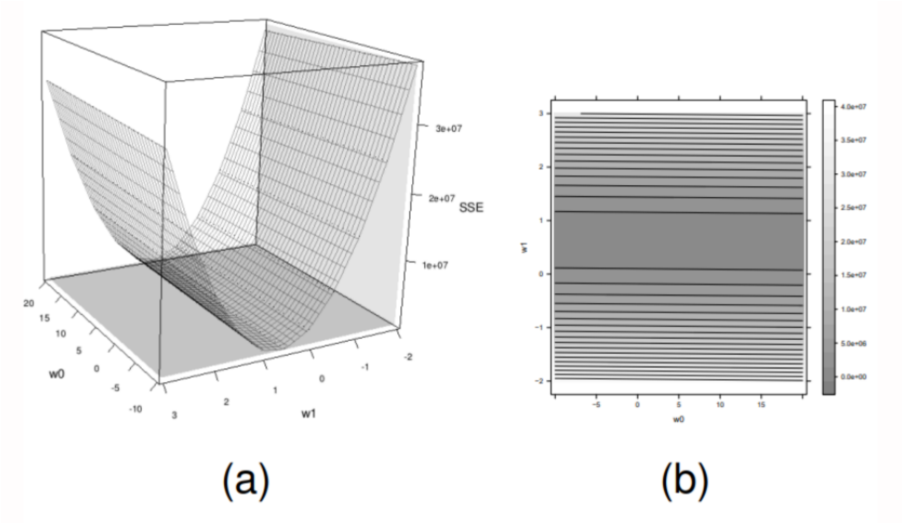


Figure 34: Error surfaces

9.2.4 Developing a Multivariate Model & Handling Categorical Features

The basic structure of the multivariable linear regression model allows for only continuous descriptive features, so we need a way to handle categorical descriptive features. THe mose common approach to handling categorical features uses a transformation which converts a single categorical descriptive feature into a number of continuous descriptive feature values that can encode the levels of the categorical feature. An example mutlivariate linear regression model might look like:

$$\begin{aligned}\textsc{rental price} = w[0] &+ w[1] \times \textsc{size}\\ &+ w[2] \times \textsc{floor}\\ &+ w[3] \times \textsc{broadband rate}\\ &+ w[4] \times \textsc{energy rating a}\\ &+ w[5] \times \textsc{energy rating b}\\ &+ w[5] \times \textsc{energy rating c}\end{aligned}$$

9.3 Evaluating Regression Models

In this section, we will introduce some of the most common performance measures used for regression tasks. Domain-specific measures of error include: mean squared error (MSE), root mean squared error (RMSE), & mean absolute error (MAE). Domain-independent measures of error include R^2. The basic evaluation process is the same as for categorical targets / classification tasks: maintain separate training & test sets (i.e., using cross validation), train the regression model on the training set, and compute the performance measures of interest on both training & test sets.

9.3.1 Mean Squarer Error (MSE)

$$\text{MSE} = \frac{\sum_{i=1}^{n} (t_i - \mathbb{M}(d_i))^2}{n}$$

where $\mathbb{M}(d_1) \dots \mathbb{M}(d_n)$ is a set of n values predicted by the model and $t_1 \dots t_n$ is a set of labels.

The MSE performance captures the average difference between the expected target values in the test set and the values predicted by the model. MSE allows us to rank the performance of multiple models on a regression problem. MSE values fall in the range $[0, \infty]$ where smaller values indicate a better model performance. However, MSE values are not especially meaningful: there is no sense of how much error occurs on individual predictions due to the squared term.

9.3.2 Root Mean Squared Error (RMSE)

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n} \left(t_i - \mathbb{M}(d_i)\right)^2}{n}}$$

where $\mathbb{M}(d_1) \ldots \mathbb{M}(d_n)$ is a set of n values predicted by the model and $t_1 \ldots t_n$ is a set of labels.

RMSE values are in the same units as the target value and thus allow us to say something more meaningful about what the error for predictions made by the model will be. RMSE values can be thought of as the "average" error on each prediction made by a regression model. Due to the inclusion of the squared term, the root mean squared error tends to overestimate error slightly as it over-emphasises individual large errors.

9.3.3 Mean Absolute Error

$$\text{MAE} = \frac{\sum_{i=1}^{n} \text{abs}\left(t_i - \mathbb{M}(d_i)\right)}{n}$$

An alternative measure that addressed the problem of large errors dominating the RMSE metric is the **mean absolute error (MAE)** which does not include a squared term. abs() in the above equation refers to the absolute value, and all other terms have the same meaning as before. As with RMSE, MAE values are in the same units as the target variable so we can say that MAE values give an indication of the "average" error on each prediction.

9.3.4 R^2

RMSE & MAE give errors that are in the same units as the target variable, which is attractive as they give an intuitive measure of how a model is performing; however, RMSE & MAE values are not sufficient to judge whether a model is making accurate predictions without having a deep knowledge of the domain (e.g., "is an error of 1.38mg acceptable?"). To make such judgements without deep domain knowledge, a normalise **domain-independent** measure of error is helpful.

The R^2 **coefficient** is a domain-independent measure that compares the performance of a model on a test set with the performance of an imaginary model that always predicts the average values from the test set. R^2 values may be interpreted as the amount of variation in the target feature that is explained by the descriptive features in the model.

$$\begin{aligned}
\text{sum of squared errors} =& \frac{1}{2} \sum_{i=1}^{n} \left(t_i - \mathbb{M}\left(d_i\right)\right)^2 \\
\text{total sum of squares} =& \frac{1}{2} \sum_{i=1}^{n} \left(t_i - \bar{t}\right)^2 \\
R^2 =& \frac{\text{sum of squared errors}}{\text{total sum of squares}}
\end{aligned}$$

where $\bar{t}$ is the average value of the target variable.

R^2 values are usually in the range $[0, 1]$, with larger values indicating better performance. However, R^2 values can be < 0 in certain rare cases (although 1 is always the maximum R^2 value). Negative R^2 values indicate a very poor model performance, i.e. that the model performs worse than the horizontal straight-line hypothesis that always predicts the average value of the target feature. For example, a negative R^2 on the test set with a positive R^2 value on the training set likely indicates that the model is overfit to the training data.

9.4 Applying k-NN to Regression Tasks

Previously, we have seen that the k-nearest neighbours algorithm bases its prediction on several (k) nearest neighbours by computing the distance from the query case to all stored cases, and picking the k nearest neighbours. When k-NN is used for classification tasks, the neighbours vote on the classification of the test case. In **regression** tasks, the average value of the neighbours is taken as the label for the query case.

9.4.1 Uniform Weighting

Assuming that each neighbour is given an equal weighting:

$$\text{prediction}(q) = \frac{1}{k} \sum_{i=1}^{k} t_i$$

where q is a vector containing the attribute values for the query instance, k is the number of neighbours, t_i is the target value of neighbour i.

9.4.2 Distance Weighting

Assuming that each neighbour is given a weight based on the inverse square of its distance from the query instance:

$$\text{prediction}(q) = \frac{\sum_{i=1}^{k} \left(\frac{1}{\text{dist}(q,d_i)^2 \times t_i} \right)}{\sum_{i=1}^{k} \left(\frac{1}{\text{dist}(q,d_i)^2} \right)}$$

where q is a vector containing the attribute values for the query instance and $\text{dist}(q, d_i)$ returns the distance between the query and the neighbour i.

9.5 Applying Decision Trees to Regression

Regression trees are constructed similarly to those for classification; the main change is that the function used to measure the quality of a split is changed so that it is a measure relevant to regression, e.g. variance, MSE, MAE, etc. This adapation is easily made to the ID3/C4.5 algorithm.

The aim in regression trees is to group similar target values together at a leaf node. Typically, a regression tree returns the mean target value at a leaf node.

10 Clustering

Heretofore, we have mainly looked only at supervised learning tasks where we have labelled data giving ground truths that we can compare predictions against. In **unsupervised learning**, there are no labels. Our goal in unsupervised learning is to develop models based on the underlying structure within the descriptive features in a dataset. This structure is typically captures in new generated features that can be appended to the original dataset to *augment* or *enrich* it.

- **Supervised learning** is task-drive with pre-categorised data, with the objective of creating predictive models. Examples include the classification task of fraud detection, and the regression task of market forecasting.
- **Unsupervised learning** is data-driven with unlabelled data, with the objective of recognising patterns in the data. Examples include the classification task of targeted marketing and the **association** task of customer recommendations.

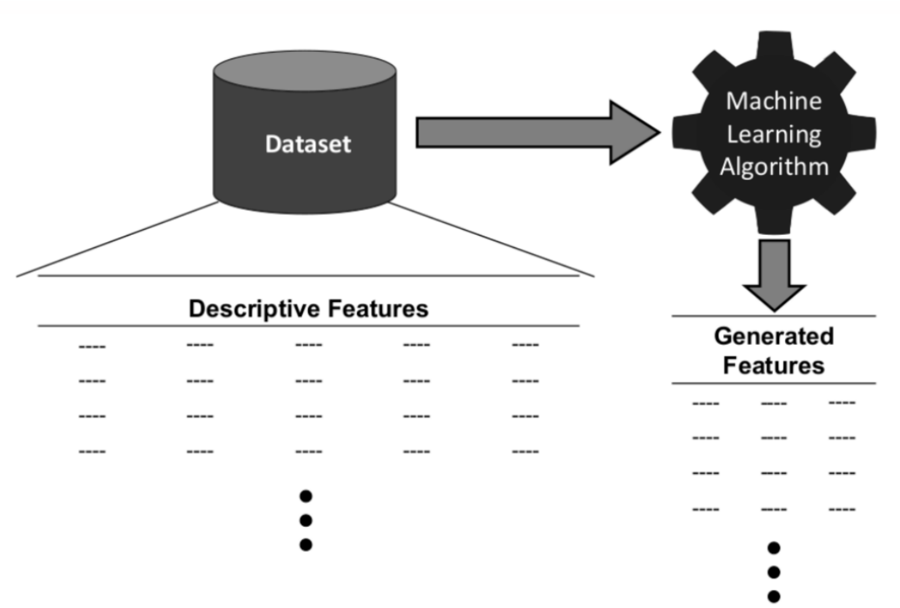


Figure 35: Unsupervised learning overview

10.1 Unsupervised Learning Task Examples

Clustering partitions the instances in a dataset into groups or *clusters* that are similar to each other. The end result of clustering is a single new generated feature that indicates the cluster than an instance belongs to, and the generation of this new feature is typically the end goal of the clustering task. Common applications of clustering include customer segmentation with which organisations attempt to discover meaningful groupings into which they can group their customers so that targeted offers or treatments can be designed. Clustering is also commonly used in the domain of information retrieval to improve the efficiency of the retrieval process: documents that are associated with each other are assigned to the same cluster.

In **representation learning**, the goal is to create a new way to represent the instances in a dataset, usually with the expectation that this new representation will be more useful for a later, usually supervised, machine learning process. It is usually achieved using specific types of deep learning models called **auto-encoders**; this is an advanced topic and so we will not discuss it in detail in this module.

10.1.1 Clustering Example: How to organise letters?

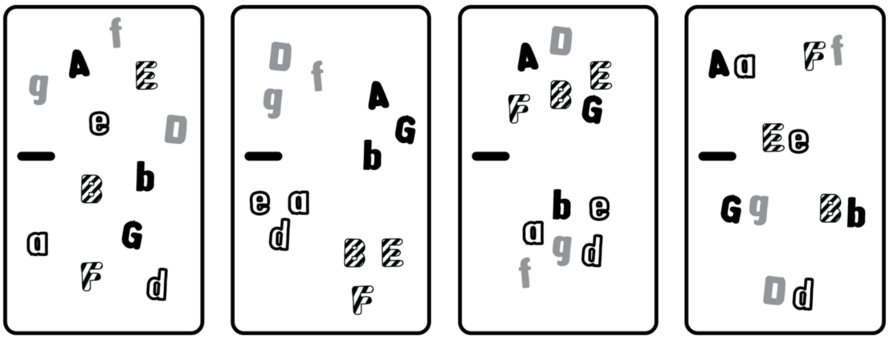


Figure 36: Clustering example: how to organise the letters?

The letters can be grouped by different features, e.g., colour, case, or character. Is there a “correct” grouping? We don’t have a ground truth available: all groupings are valid in this case as they each highlight different characteristics present in the dataset.

10.2 Clustering using the k-Means Algorithm

We have already covered many of the fundamentals required to tackle clustering problems, e.g., feature spaces & measuring similarity using distance metrics (as we did with k-nearest neighbours). The **k-means** clustering algorithm is the most well-known approach to clustering. It is:

- Relatively easy to understand.
- Computationally efficient.
- Simple to implement but also usually very effective.

$$\sum_{i=1}^{n} \min_{c_1,\ldots,c_k} \text{dist}(d_i, c_j)$$

Given a dataset D consisting of n instances $d_1 \ldots d_n$ where d_i is a set of m descriptive features:

- The goal when applying k-means is to divide this dataset into k disjoint clusters $C_1 \ldots C_k$.
- The number of clusters k is an input to the algorithm.
- The division into clusters is achieved by minimising the result of the above equation.
- $c_1 \ldots c_k$ are the centroids of the clusters; they are vectors containing the co-ordinates in the feature space of each cluster centroid.
- dist()is a distance metric (defined the same way as we previously defined distance metrics when studying k-NN), i.e., a way to measure the similarity between instances in a dataset.

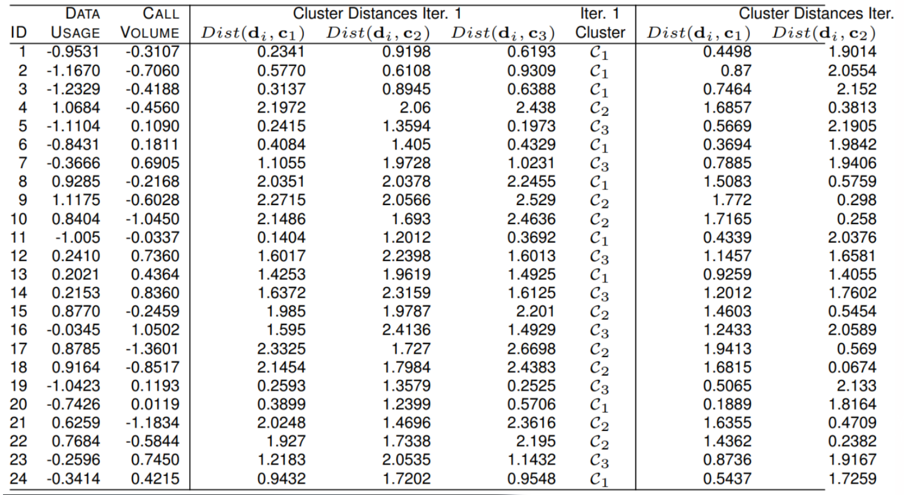
Algorithm 2 Pseudocode description of the k-means clustering algorithm

```
Require: A dataset $D$ containing $n$ training instances, $d_1, \ldots, d_n$
Require: The number of clusters to find $k$
Require: A distance measure, dist(), to compare instances to cluster centroids
Select $k$ random cluster centroids, $c_1$ to $c_k$, each defined by values for each descriptive feature, $c_i = \langle c_i[1], \ldots, c_i[m] \rangle$
repeat
    for each instance $d_i$ do
        Calculate the distance of $d_i$ to each cluster centroid $c_1$ to $c_k$, using $Dist$
        Assign $d_i$ to the cluster $C_i$ whose centroid $c_i$ it is closest to
    end for
    for each cluster $C_i$ do
        Update the centroid $c_i$ to the average of the descriptive feature values of instances in $C_i$
    end for
until no cluster reassignments are performed during an iteration
```

Issues to consider when applying k-means include:

- **Choice of distance metric:** it's common to use Euclidean distance. Other distance metrics are possible, but may break convergence guarantees. Other clustering algorithms, e.g., k-medoids, have been developed to address this problem.
- **Normalisation of data:** as we are measuring similarity using distance, as with k-NN, normalising the data beforehand is very important when the values of the attributes have different ranges, otherwise attributes with the largest ranges will dominate calculations.
- **How to identify when convergence happens?:** when no cluster memberships change during a full iteration of the algorithm.
- **How to choose a value for k?:** we will discuss this later in the coming slides.

10.2.1 Mobile Phone Customer Example Dataset



	Data	Call	Cluster Distances Iter. 1			Iter. 1	Cluster Distances Iter.	
ID	Usage	Volume	$Dist(\mathbf{d}_i, \mathbf{c}_1)$	$Dist(\mathbf{d}_i, \mathbf{c}_2)$	$Dist(\mathbf{d}_i, \mathbf{c}_3)$	Cluster	$Dist(\mathbf{d}_i, \mathbf{c}_1)$	$Dist(\mathbf{d}_i, \mathbf{c}_2)$
1	-0.9531	-0.3107	0.2341	0.9198	0.6193	$\mathcal{C}_1$	0.4498	1.9014
2	-1.1670	-0.7060	0.5770	0.6108	0.9309	$\mathcal{C}_1$	0.87	2.0554
3	-1.2329	-0.4188	0.3137	0.8945	0.6388	$\mathcal{C}_1$	0.7464	2.152
4	1.0684	-0.4560	2.1972	2.06	2.438	$\mathcal{C}_2$	1.6857	0.3813
5	-1.1104	0.1090	0.2415	1.3594	0.1973	$\mathcal{C}_3$	0.5669	2.1905
6	-0.8431	0.1811	0.4084	1.405	0.4329	$\mathcal{C}_1$	0.3694	1.9842
7	-0.3666	0.6905	1.1055	1.9728	1.0231	$\mathcal{C}_3$	0.7885	1.9406
8	0.9285	-0.2168	2.0351	2.0378	2.2455	$\mathcal{C}_1$	1.5083	0.5759
9	1.1175	-0.6028	2.2715	2.0566	2.529	$\mathcal{C}_2$	1.772	0.298
10	0.8404	-1.0450	2.1486	1.693	2.4636	$\mathcal{C}_2$	1.7165	0.258
11	-1.005	-0.0337	0.1404	1.2012	0.3692	$\mathcal{C}_1$	0.4339	2.0376
12	0.2410	0.7360	1.6017	2.2398	1.6013	$\mathcal{C}_3$	1.1457	1.6581
13	0.2021	0.4364	1.4253	1.9619	1.4925	$\mathcal{C}_1$	0.9259	1.4055
14	0.2153	0.8360	1.6372	2.3159	1.6125	$\mathcal{C}_3$	1.2012	1.7602
15	0.8770	-0.2459	1.985	1.9787	2.201	$\mathcal{C}_2$	1.4603	0.5454
16	-0.0345	1.0502	1.595	2.4136	1.4929	$\mathcal{C}_3$	1.2433	2.0589
17	0.8785	-1.3601	2.3325	1.727	2.6698	$\mathcal{C}_2$	1.9413	0.569
18	0.9164	-0.8517	2.1454	1.7984	2.4383	$\mathcal{C}_2$	1.6815	0.0674
19	-1.0423	0.1193	0.2593	1.3579	0.2525	$\mathcal{C}_3$	0.5065	2.133
20	-0.7426	0.0119	0.3899	1.2399	0.5706	$\mathcal{C}_1$	0.1889	1.8164
21	0.6259	-1.1834	2.0248	1.4696	2.3616	$\mathcal{C}_2$	1.6355	0.4709
22	0.7684	-0.5844	1.927	1.7338	2.195	$\mathcal{C}_2$	1.4362	0.2382
23	-0.2596	0.7450	1.2183	2.0535	1.1432	$\mathcal{C}_3$	0.8736	1.9167
24	-0.3414	0.4215	0.9432	1.7202	0.9548	$\mathcal{C}_1$	0.5437	1.7259

Figure 37: Example of normalised mobile phone customer dataset with the first 2 iterations of k-means

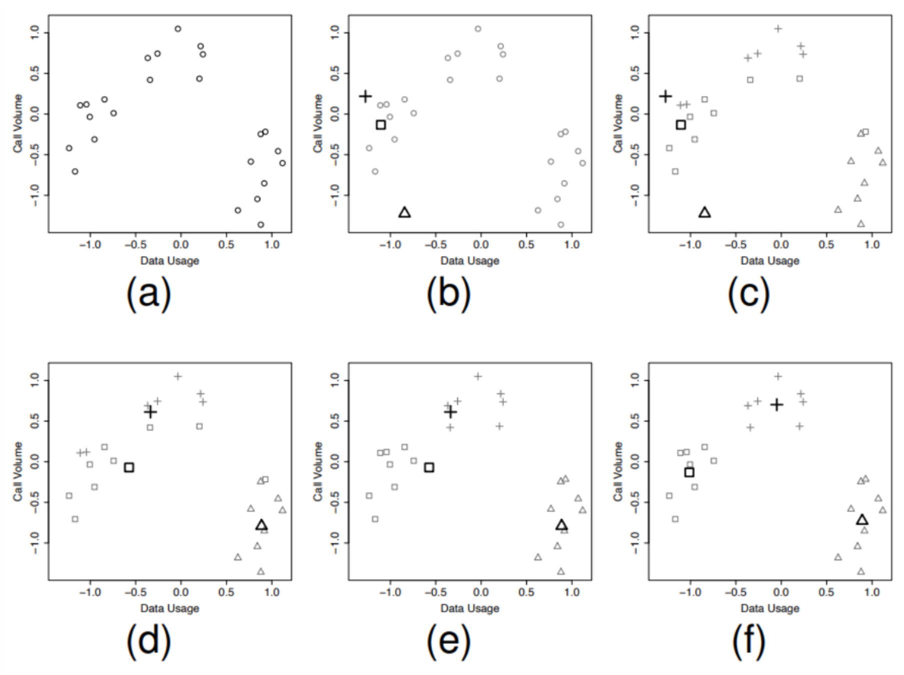


Figure 38: Plot of the mobile phone customer dataset

Large symbols represent clusters, while small symbols represent individual data points. It is clear to a human viewer that there are 3 natural clusters within the dataset, but we require an algorithm such as k-means clustering to find them automatically. Generally, it is not possible to determine the correct number of clusters by eye in real word high-dimensional datasets.

Each cluster centroid is then updated by calculating the mean value of each descriptive feature for all instances that are a member of the cluster.

$$
\begin{aligned}
c_1[\textsc{Data Usage}] =& \frac{(-0.9531 + -1.167 + -1.2329 + -0.8431 + 0.9285 + -1.005 + 0.2021 + -0.7426 + -0.3414)}{9} \\
=& -0.5727 \\
c_1[\textsc{Call Volume}] =& \frac{(-0.3107 + -0.706 + -0.4188 + 0.1811 + -0.2168 + -0.0337 + 0.4364 + 0.0119 + 0.4215)}{9} \\
=& -0.0706
\end{aligned}
$$

Once the algorithm has completed, its two outputs are a vector of assignments of each instance in the dataset to one of the clusters $C_1 \ldots C_k$ and the k cluster centroids $c_1 \ldots c_k$. The assignment of instances to clusters can then be used to enrich the original dataset with a new generated feature: the cluster memberships.

$$
\begin{aligned}
C_1 =& \{d_1, d_2, d_3, d_5, d_6, d_{11}, d_{19}, d_{20}\} \\
C_2 =& \{d_4, d_8, d_9, d_{10}, d_{15}, d_{17}, d_{18}, d_{21}, d_{22}\} \\
C_3 =& \{d_7, d_{12}, d_{13}, d_{14}, d_{16}, d_{23}, d_{24}
\end{aligned}
$$

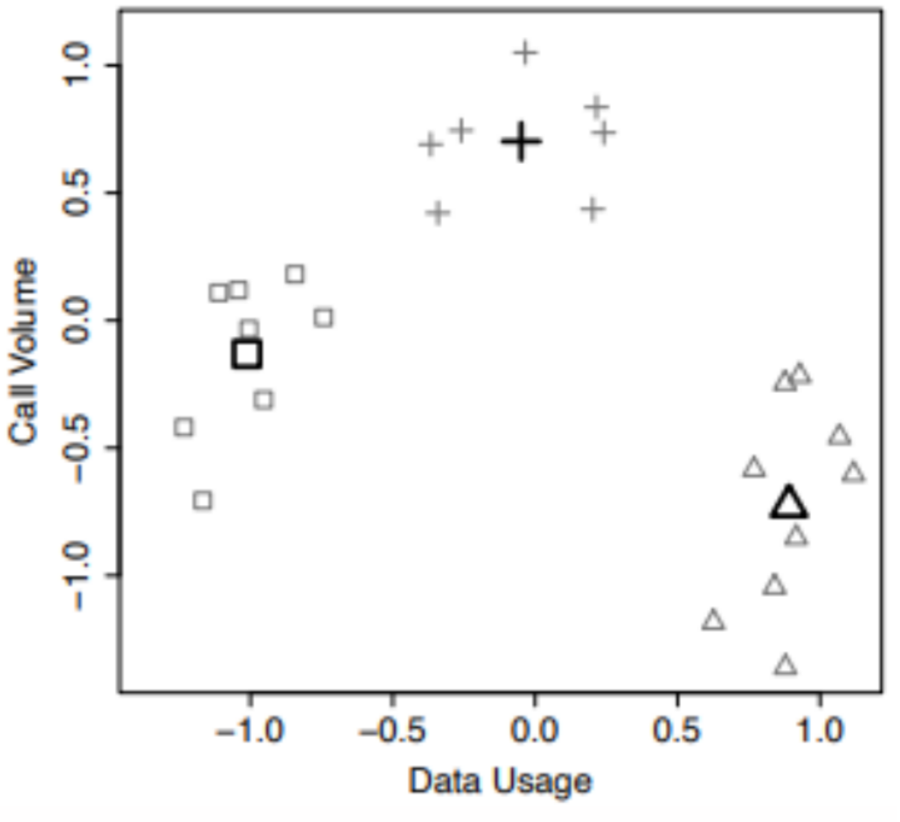
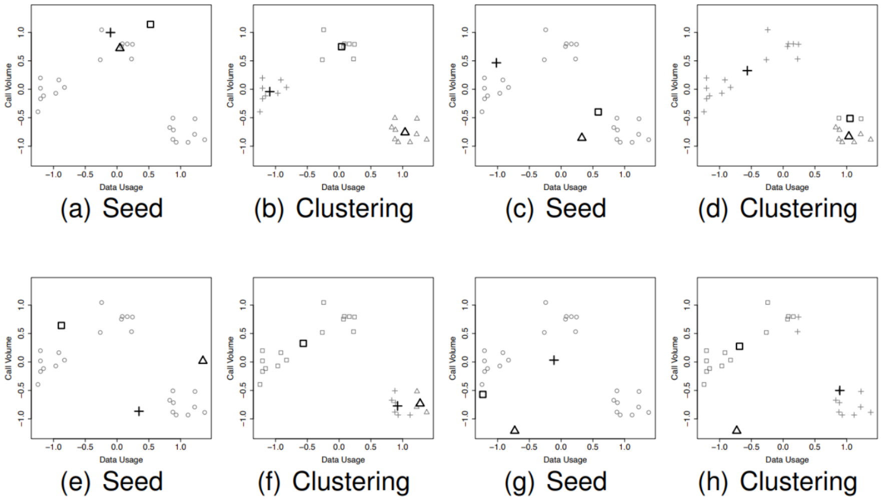


Figure 39: Final k-means clustering

10.3 Choosing Initial Cluster Centroids

How should we choose the initial set of cluster centroids? We could choose k random initial cluster centroids as per the pseudocode. The choice of these initial cluster centroids (seeds), unfortunately, can have a big impact on the performance of the algorithm. Different randomly selected starting points can lead to different, often sub-optimal, clusterings (i.e., local minima). As we will see in the following example, a particularly unlucky clustering could lead to a cluster having no members upon convergence. This has an easy fix: choose random instances in the dataset as the seeds. An easy way to address this issue is to perform multiple runs of the k-means clustering algorithm starting from different initial centroids and then aggregate the results. In this way, the most common clustering is chosen as the final result.



(a) Seed (b) Clustering (c) Seed (d) Clustering

(e) Seed (f) Clustering (g) Seed (h) Clustering

Figure 40: The effect of different seeds on final clusters

10.3.1 k-means++

Our main goals when selecting the initial centroids is to find initial centroids less likely to lead to sub-optimal clusterings and to select initial centroids that allow the algorithm to converge much more quickly than when seeds are randomly chosen.

The default method for this in scikit-learn is k-means++: in this approach, an instance is chosen randomly (following a uniform distribution) from the dataset as the first centroid. Subsequent centroids are then chosen randomly but following a distribution defined by the square of the distances between an instance and the nearest cluster centroid out of those found so far. This means that instances far away from the current set of centroids are much more likely to be selected than those close to already selected centroids.

As before, we are using the mobile phone customers dataset with $k = 3$. Instances from the dataset are being used as cluster centroids. Typically, there is good diversity across the feature space in the centroids selected. The k-means++ algorithm is still stochastic: it does not completely remove the possibility of a poor starting point that leads to a sub-optimal clustering. Therefore, we should still try running it multiple times and pick the most common clustering.

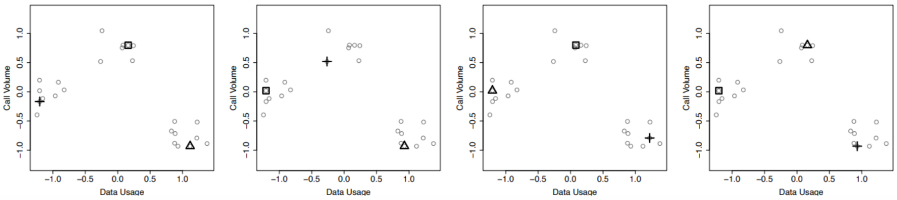


Figure 41: Examples of initial seeds chosen by k-means++

Algorithm 3 Pseudocode description of the k-means++ algorithm

```
Require: A dataset $D$ containing $n$ training instances, $d_1, \ldots, d_n$
Require: $k$, the number of cluster centroids to find
Require: A distance measure $Dist$ to compare instances to cluster centroids
Choose $d_i$ randomly (following a uniform distribution) from $D$ to be the position of the initial centroid, $c_1$, of the first cluster, $C_1$
for cluster $C_j$ in $C_2$ to $C_k$ do
    for each instance $d_i$ in $D$ do
        Let $Dist(d_i)$ be the distance between $d_i$ and its nearest cluster centroid
    end for
    Calculate a selection weight for each instance $d_i$ in $D$ as
        $$\frac{Dist(d_i)^2}{\sum_{p=1}^{n} Dist(d_p)^2}$$
    Choose $d_i$ as the position of cluster centroid $c_j$ for cluster $C_j$ randomly following a distribution based on the selection weights
end for
Proceed with k-means as normal using $\{c_1, \ldots, c_k\}$ as the initial centroids.
```

10.4 Evaluating Clustering

Evaluation is more complicated when conducting unsupervised learning compared to supervised learning. All performance measures that we have previously discussed for supervised learning relied on having **ground truth** labels available, which we can use to objectively measure performance using metrics like accuracy, error rate, RMSE, MAE, etc.

We could use an idealised notion of what a "good" clustering looks like, i.e. instances in the same cluster should all be relatively close together and instances belonging to different clusters should be far apart.

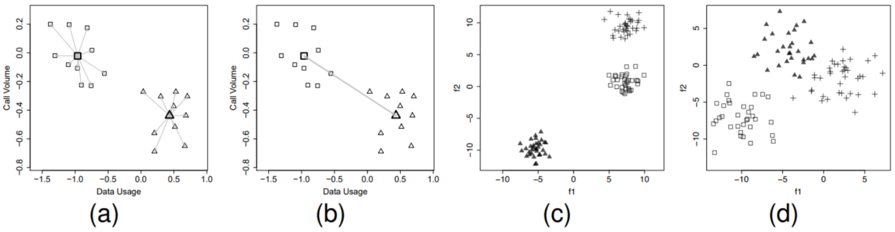


Figure 42: (a) Intra-cluster distance, (b) inter-cluster distance, (c) a good clustering, (d) a bad clustering

10.4.1 Loss function for clustering

A **loss function** is a function we wish to minimise to reduce error on a ML model. **Inertia** is the sum of the squared distances from each data point to its centre; it is alternatively referred to as the sum of squared errors (SSE). The "error" in this case is the distance between the centroid of the cluster and each data point in that cluster. To calculate, simply determine all distances from each data point to the centroid of its assigned cluster, square the distances, and sum them up. We can compare different clusterings using inertia values.

10.4.2 Using the Elbow Method to Determine k

Plot the inertia/SSE on the y-axis against the number of clusters k on the x-axis. Other more complex methods are available, e.g., silhouette which we will not cover in this module.

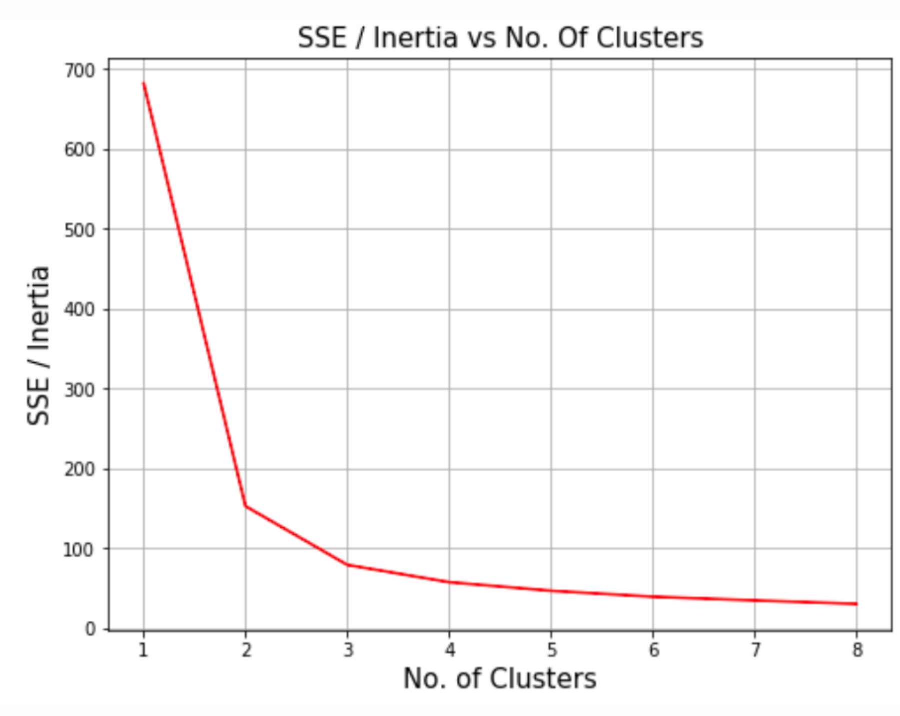


Figure 43: The elbow method gives an indication of an appropriate k-value — in this case, $k = 3$ looks appropriate.

Note that no definitive methods are available to determine the correct number of clusters. We may need to be informed by deep knowledge of the problem domain.

10.5 Picking k: Real-World Example

Sometimes, we can rely on *domain-specific metrics* to guide the choice of k. For example:

- Cluster heights & weights of customers with $k = 3$ to design small, medium, & large shirts.
- Cluster heights & weights of customers with $k = 5$ to design XS, S, M, L, & XL shirts.

To pick k in the examples above, consider the projected costs and sales for the two different k values and pick the value of k that maximises profit.

10.6 Hierarchical Clustering

Hierarchical clustering (HC) is a general family of clustering algorithms that build nested clusters by merging or splitting them successively. This hierarchy of clusters is represented as a tree or **dendrogram**. The root of the tree is the unique cluster that gathers all the samples, and the leaves are the clusters that contain only one sample. Hierarchical clustering methods can sometimes by more computationally expensive than simpler algorithms (e.g., k-means).

Agglomerative hierarchical clustering (AHC) is an example of a hierarchical clustering method. It starts by considering each instance in the dataset as a member of its own individual cluster, then merging the most similar adjacent clusters in each iteration. It can find clusters that are not possible to find with k-means. We will not cover this algorithm in detail in this module, just illustrate key differences with results given by k-means.

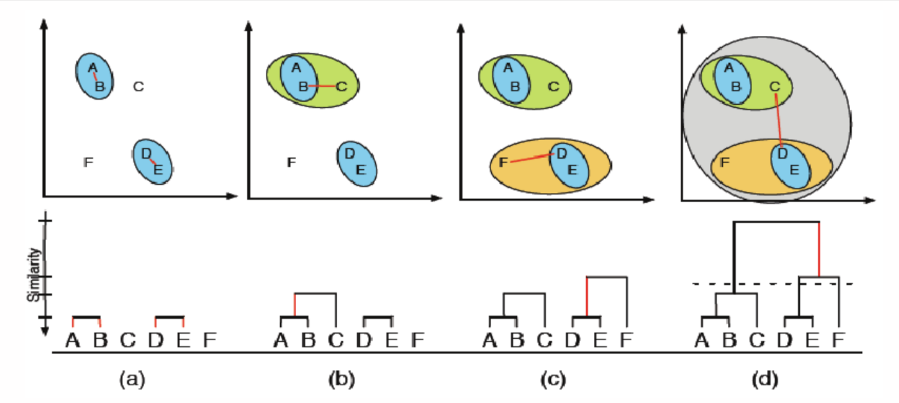
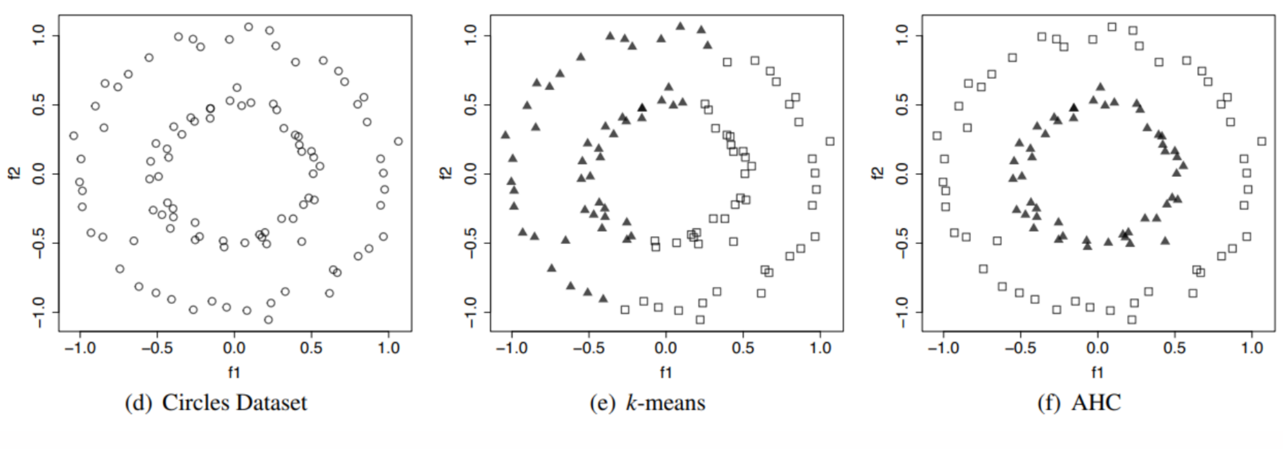


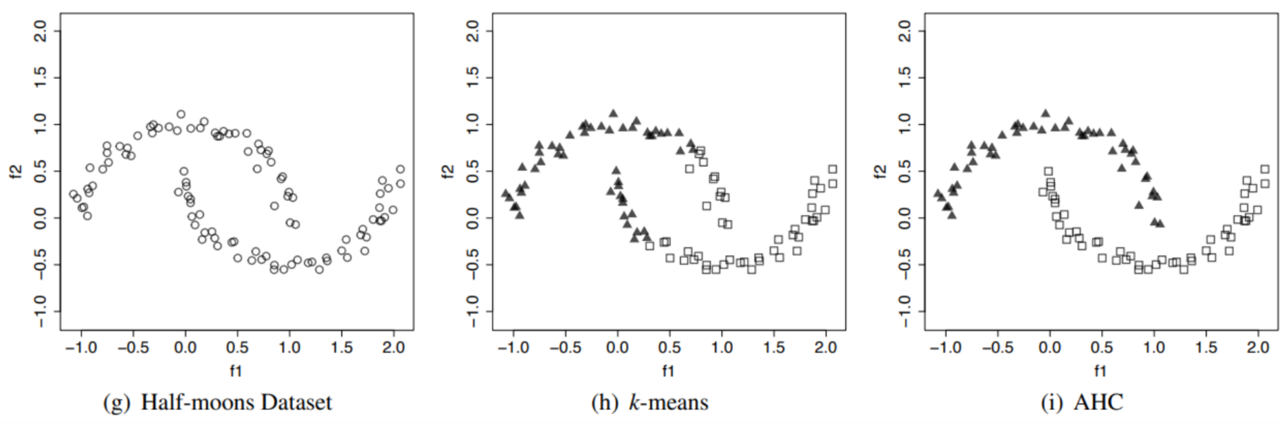
Figure 44: Agglomerative Hierarchical Clustering

k-means cannot find the clusters that we might expect when "eyeballing" the plots, due to the underlying assumptions that k-means makes about how the points in a cluster should be distributed.



(d) Circles Dataset (e) k-means (f) AHC

Figure 45: k-means vs AHC on circles dataset



(g) Half-moons Dataset (h) k-means (i) AHC

Figure 46: k-means vs AHC on half-moons dataset

11 Neural Networks

11.1 Introduction to Deep Learning

Deep learning is an approach to machine learning that is inspired by how the brain is structured & operates. It is a relatively new term that describes research on modern artificial neural networks (ANNs).

Artificial neural network models are composed of large numbers of simple processing units called **neurons** that typically are arranged into layers and are highly interconnected. Artificial neural networks are some of the most powerful machine learning models, able to learn complex non-linear mappings from inputs to outputs. ANNs generally work well in domains in which there are large numbers of input features (such as image, speech, or language processing), and for which there are very large datasets available for training.

The history of ANNs dates to the 1940s. The term “deep learning” became prominent in the mid-200s. The term “deep learning” emphasises that modern networks are deeper (in terms of *number of layers* of neurons) than previous networks. This extra depth enables the networks to learn more complex input-output mappings.

11.2 Artificial Neurons

The fundamental building block of a neural network is a computational model known as an **artificial neuron**. They were first defined by McCulloch & Pitts in 1943 who were trying to develop a model of the activity in the human brain based on propositional logic. They recognised that propositional logic using a Boolean representation and neurons in the brain are similar, as they have an all-or-none character (i.e., they act as a switch that responds to a set of inputs by outputting either a high activation or no activation). They designed a computational model of the neuron that would take in multiple inputs and then output either a high signal (1) or a low signal (0). The McCulloch & Pitts model has a two-part structure:

1. Calculation of the result of a **weighted sum** (we refer to the result as z). In the first stage of the McCulloch & Pitts model, each input is multiplied by a weight, and the results of these multiplications are then added together. This calculation is known as a weighted sum because it involves summing the weighted inputs to the neuron.

2. Passing the result of the weighted sum through a **threshold activation function**.

11.2.1 Weighted Sum Calculation

$$\begin{aligned}
z =& w[0] \times d[0] + \cdots + w[m] \times d[m] \\
=& \sum_{j=0}^{m} w[j] \times d[j] \\
=& w \cdot d \\
=& w^{\mathrm{T}} d = [w_0, \ldots, w_m] \begin{bmatrix} d_0 \\ \vdots \\ d_m \end{bmatrix}
\end{aligned}$$

where d is a vector of size $m + 1$ inputs / descriptive features and $d[0]$ is dummy feature that is always equal to 1. w is a vector of $m + 1$ weight, one weight for each feature and $w[0]$ is the weight for the dummy feature. Note the similarity to the linear regression equation, although (networks of) artificial neurons can do much more than just tackle regression tasks.

11.2.2 Weights

The weights can either be:

- **excitatory:** having a positive value which increases the probability of the neuron activating or;

- **inhibitory:** having a negative value, which decreases the probability of a neuron firing.

$w[0]$ is the equivalent of the y-intercept in the standard equation of a line in that the neuron that this weight captures is constantly effecting. This $w[0]$ term is often referred to as the **bias** parameter because in the absence of any other input, the output of the weighted sum is biased to be the value of $w[0]$. Technically, the inclusion of the bias parameter as an extra weight in this operation changes the function from a linear function on the inputs to an **affine** function: a function that is composed of a linear function followed by a translation (i.e., the inclusion of the bias term means that the straight line would not pass through the origin).

$d[0]$ is a dummy feature used for notational convenience that is always equal to 1.

11.2.3 Artificial Neuron Schematic

Arrows carry activations in the direction the arrow is pointing. The weight label on each arrow represents the weight that will be applied to the input carried along the arrow. ϕ is the activation function of the neuron, also known as the perceptron.

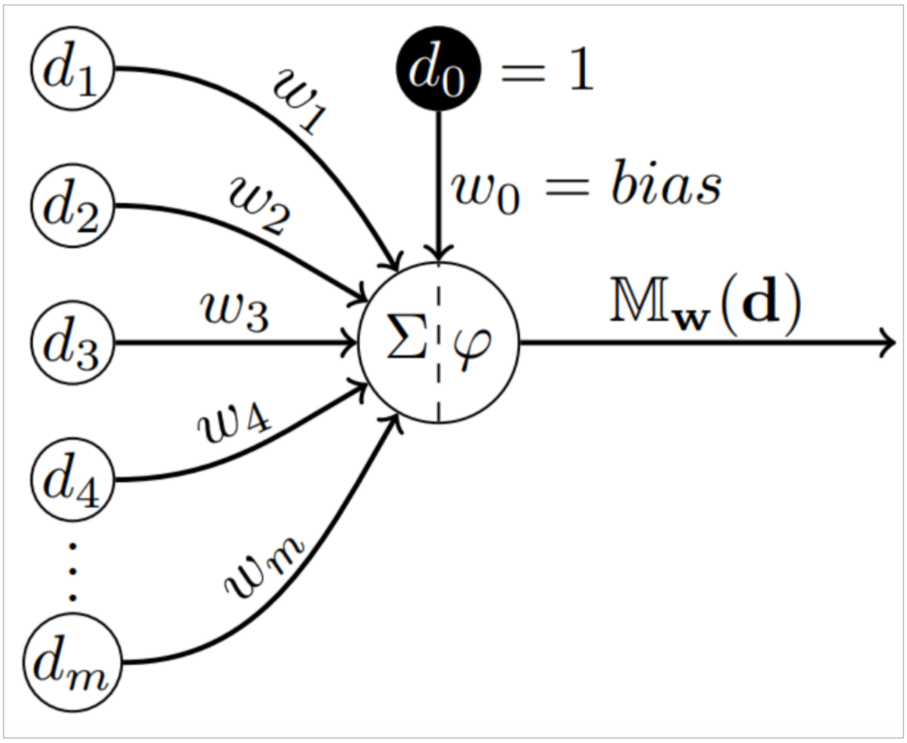


Figure 47: Artificial neuron schematic

The two-part structure of the McCulloch & Pitts model is the basic blueprint for modern artificial neurons. The key differences are:

- In the McCulloch & Pitts model, the weights were manually set (very difficult to do); in modern machine learning, we learn the weights using data.
- McCulloch & Pits considered the threshold activation function only; modern artificial neurons use one of a range of different activation functions.

11.3 Threshold Activation Function

In the second stage of the McCulloch & Pitts model, the result of the weighted sum calculation z is then converted into a high or low activation by passing z through the **activation function**.

McCulloch & Pitts used a **threshold activation function:** if z is greater than or equal to the threshold, the artificial neuron outputs a 1 (high activation), otherwise it outputs a 0 (low activation). Using the symbol θ to denote the threshold, the second stage of processing in the McCulloch & Pitts model can be defined. We'll see that the neuron "fires" when $z \geq \theta$.

$$\mathbb{M}_w(d) = \begin{cases} 1 & \text{if } z \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

ϕ is the **activation function** of the neuron.

$$\begin{aligned}\mathbb{M}_w(d) =& \phi(w[0] \times d[0] + \cdots + w[m] \times d[m]) \\ =& \phi\left(\sum_{i=0}^{m} w_i \times d_i\right) = \phi\left(w \cdot d\right) \\ =& \phi\left(w^{\mathrm{T}} d\right)\end{aligned}$$

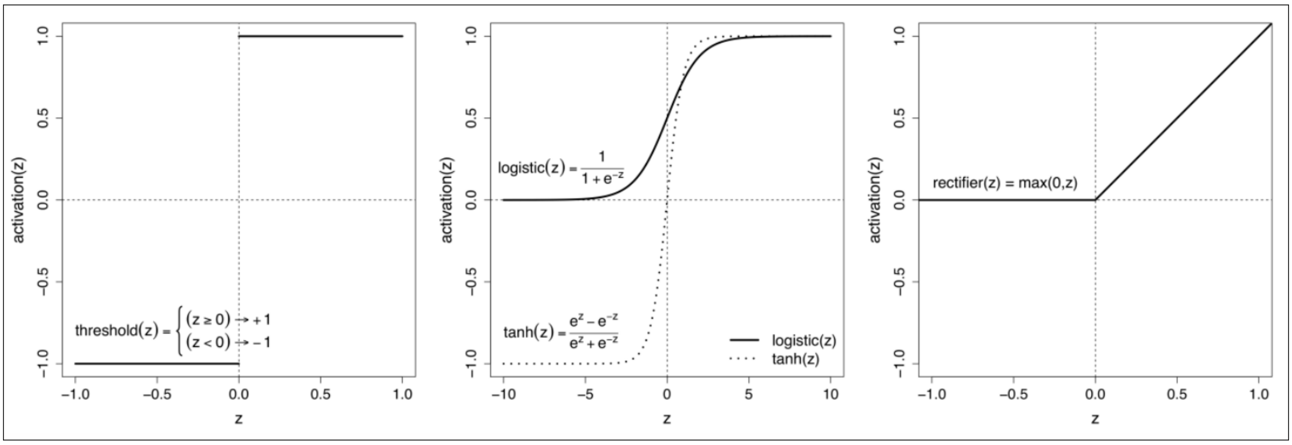


Figure 48: Example **non-linear** activation functions

11.3.1 Logistic Activation Function

Historically, the **logistic activation function** was among the most commonly-used choices of activation functions. Note that neurons are often referred to as *units*, and they are distinguished by the type of activation function that they use; hence, a neuron that uses a logistic activation function is referred to as a **logistic unit**. Logistic activation functions may map any real number input to an output between 0 & 1.

$$\text{logistic}(z) = \frac{1}{1 + e^{-z}}$$

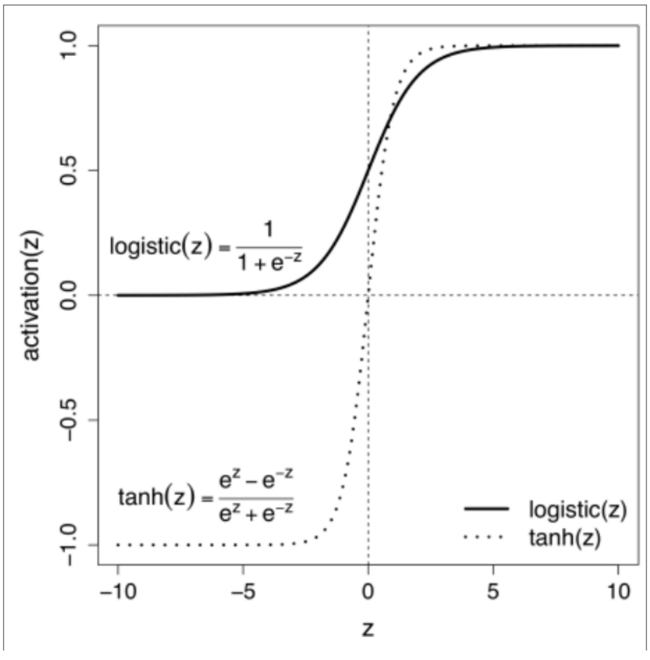


Figure 49: Logistic activation function

11.3.2 Rectified Linear Activation Function

Today, the most popular choice of activation function is the **rectified linear activation** function or **rectifier**. A unit that uses the rectifier function is known as a **rectified linear unit** or **ReLU**. The rectified linear function does not *saturate*, i.e., there is no maximum output value constraint.

$$\text{rectifier}(z) = \max(0, z)$$

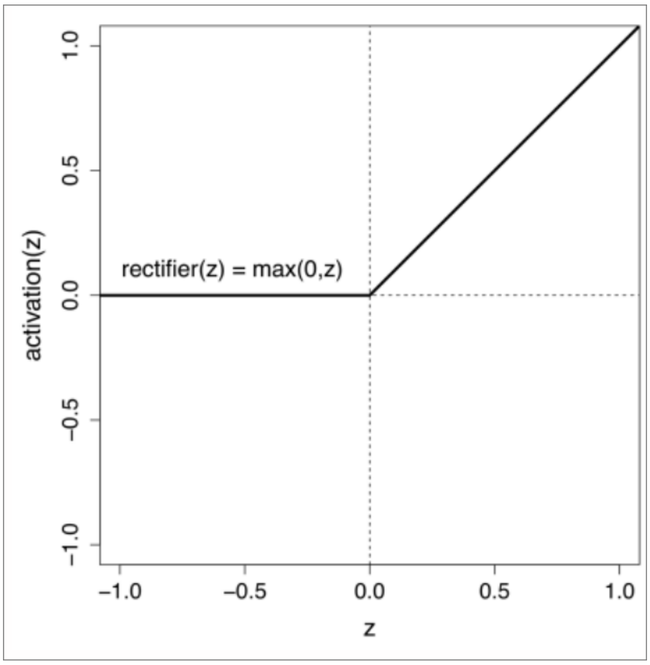


Figure 50: Rectified linear activation function

11.4 Artificial Neural Networks

An **artificial neural network** (also called a *multi-layer perceptron* or MLP) consists of a networks of inter-connected artificial neurons.

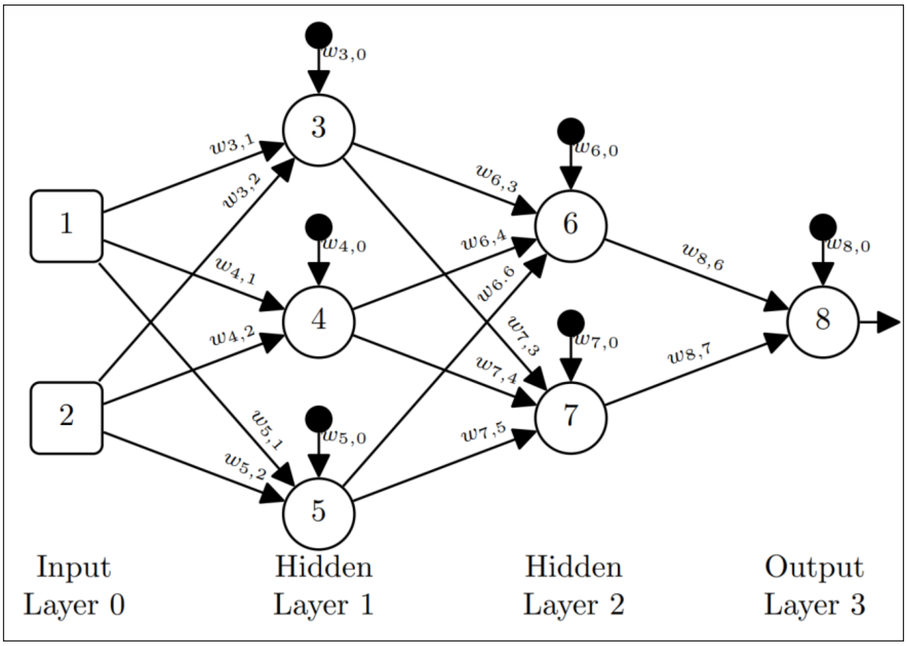


Figure 51: A **feedfoward** neural network. The neurons are organised into a sequence of *layers*.

There are two types of neurons in this network: *sensing neurons* & *processing neurons*. The two squares on the left of the figure represent the two memory locations through which inputs are presented to this network. These locations can be thought of as *sensing* neurons that permit the network to sense the external inputs; although we consider these memory locations as sensing neurons within the network, the inputs presented to the network are not transformed by these sensing neurons.

Circles in the network represent *processing* neurons that transform input using the aforementioned two-step process of a weighted sum followed by an activation function.

Arrows connecting the neurons in the network indicate the flow information through the network.

This artificial neural network is **fully-connected** because each of the neurons in the network is connected so that it receives inputs from all the neurons in the preceding layer and passes its output activation to all the neurons in the next layer.

An artificial neural network can have any structure, but a layer-based organisation of neurons is common. Input to processing neurons can be:

- External input from sensing neurons;
- The output activation of another processing neuron in the network;
- A dummy input that is always set to 1 (the input from a black circle).

In feedfowaard networks, there are no loops or cycles in the network connections that would allow the output of a neuron to flow back into the neuron as an input (even indirectly). In a feedfoward network, the activations in the network always flow *forward* through the sequence of layers. The **depth** of a neural network is the number of hidden layers plus the output layer.

11.4.1 When is a Neural Network Considered "Deep"?

The number of layers required for a network to be considered **deep** is an open question. Cybenko (1988) proved that a network with three layers of (processing) neurons (i.e., two hidden layers & an output layer) can approximate any function to arbitrary accuracy. Therefore, we here define the minimum number of hidden layers necessary for a network to be considered deep as **two**; under this definition, the network shown previously would be described as a deep network. However, most deep networks have many more than two hidden layers: some deep networks have tens or even hundreds of layers.

11.4.2 Notes on Activation Functions

It is not a coincidence that the most useful activation functions are non-linear. Introducing non-linearity into the input-to-output mappings defined by a neuron enables an artificial neural network to learn complex non-linear mappings. This ability to learn complex non-linear mappings makes artificial neural networks such powerful models in terms of their ability to be accurate on complex tasks.

A multi-layer feedforward network that uses only linear neurons (i.e., neurons that do not include a non-linear activation function) is equivalent to a single-layer network with linear neurons; in other words, it can represent only a linear mapping on the inputs. This equivalence is true no matter how many hidden layers we introduce into the network. Introducing even simple non-linearities in the form of logistic or rectified linear units is sufficient to enable neural networks to represent arbitrarily complex functions, provided the network contains enough layers.

11.4.3 Linear Separability

All three functions shown below have the same structure: they all take two inputs that can be either TRUE (1) or FALSE (0), and they return either TRUE (1) or FALSE (0). Solid black dots represent FALSE outputs and clear dots represent the TRUE outputs.

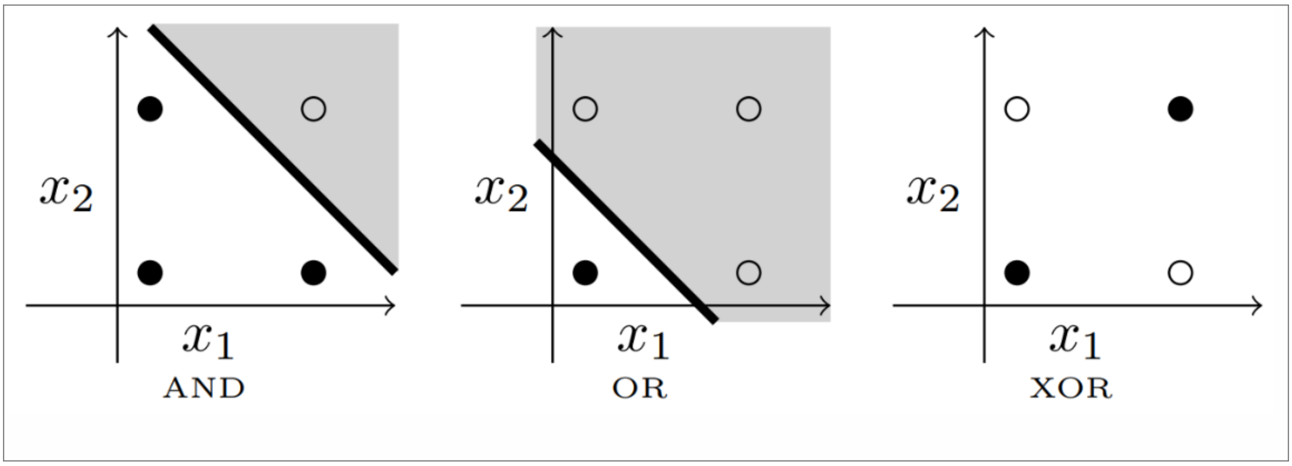


Figure 52: Example functions from Boolean logic

- AND returns TRUE if both inputs are TRUE (linearly separable);
- OR returns TRUE if either input is TRUE (linearly separable);
- XOR returns TRUE if only one of the inputs is TRUE (not linearly separable).

11.4.4 Network Depth versus Learning Rate

Adding depth to a network comes with a cost. As we will see when we discuss the *vanishing gradient problem*, adding depth to a network can slow down the rate at which a network learns. Therefore, when we wish to increase the representational power of a network, there is often a trade-off between making a network deeper and making the layers wider. Finding a good equilibrium for a given prediction task involves experimenting with different architectures to see which performs best, i.e., we can treat the neural network architecture as a hyperparameter that must be optimised. Overall, there is a general trend that deeper networks have better performance than shallower networks, but as networks become deeper, they can become more difficult to train.

11.4.5 Training Neural Networks

The **blame assignment problem** is the question of how to calculate how much of the error in the entire network is due to an individual neuron when learning neural network weights. The **backpropagation problem** solves the blame assignment problem: at each learning step, once we have used backpropagation to calculate errors for individual neurons, we can then use **gradient descent** to update the weights for each neuron in the network.

For example, consider a regression problem wherein we predict a continuous target value based on the values of two independent attributes. We could use the feedforward network structure outlined previously to tackle this. An example process could look like:

1. We initialise the weights *randomly* at first.
2. We make predictions and the use these to compute the error of the network using a **loss function**, e.g., sum of squared errors on training data.
3. We use **back propagation** to compute errors on each neuron.
4. We use **gradient descent** to update neuron weights.
5. We repeat this for multiple **epochs** (training steps) until convergence.

12 Exam Notes

- Know formula for distance formulas etc.
- Definitely know formulas for entropy and information gain – need to be able to calculate them ourselves to answer a question on decision trees.
- More or less the same format, will not try to throw in questions to catch us off-guard.
- Stick to learning objectives – questions will be related to them.
- Will focus on what we covered in the notes – no specific algorithms that we studied ourselves.
- What does it mean to have stratified samples, cross validation, etc could come up.
- Know pseudocode for all the algorithms and their associated formulas.
- For KNN, know distance formulae, could be good to know distance-weighted KNN.
- Know formulae for activation functions, how they work, and different steps.
- High level is fine for what we covered today.
- Know difference between classification and regression.
- Supervised vs unsupervised + examples.
- Describe different types of supervised vs unsupervised tasks and algorithms that might be used for them.
- What's K in k-means? Do you have to decide it in advance? How to get good estimate? Using elbow method?
- Exam format in line with previous years, but will not be cut and paste. Same types of questions in different contexts.
- Put together your own confusion matrix, true positive rate, false positive rate.
- Go back to learn objectives and tick them off.
- z-normalisation formula?
- RMSE.